Mise en oeuvre des interactions en environnements distribués, compilés et fortement typés : le modèle MICADO

Laurent Berger

Numéro d'ordre : .....

THÈSE DE DOCTORAT

École doctorale « Sciences et Technologies de l'Information et de la Communication » de Nice-Sophia Antipolis
Discipline Informatique

UNIVERSITÉ DE NICE SOPHIA-ANTIPOLIS
FACULTÉ DES SCIENCES ET TECHNIQUES

MISE EN ŒUVRE DES INTERACTIONS EN ENVIRONNEMENTS DISTRIBUÉS, COMPILÉS ET FORTEMENT TYPÉS : LE MODÈLE MICADO

---

présentée et soutenue publiquement par

Laurent BERGER

le 12 Octobre 2001, à l'ESSI, devant le jury composé de

Président du Jury	Paul FRANCHI-ZANNETTACCHI	Professeur, Université de Nice-Sophia Antipolis
Rapporteurs	Pierre COINTE	Professeur, École des Mines de Nantes (EMN)
	Philippe MERLE	Maître de Conférences, Laboratoire d'Informatique Fondamentale de Lille (LIFL)
	Christian QUEINNEC	Professeur, Université de Paris 6 (LIP 6)
Examinateurs	Sabine MOISAN	Chargée de Recherche, INRIA Sophia-Antipolis
	Peter SANDER	Professeur, Université de Nice-Sophia Antipolis
Directeur de thèse	Anne-Marie PINNA-DÉRY	Maître de Conférences, Université de Nice-Sophia Antipolis

MISE EN ŒUVRE DES INTERACTIONS EN ENVIRONNEMENTS DISTRIBUÉS, COMPILÉS ET FORTEMENT TYPÉS : LE MODÈLE MICADO

Résumé

La programmation orientée objet a déjà prouvé ses intérêts lors de la mise en œuvre d'applications complexes. Le développement des applications distribuées à l'aide de technologies objets est réalisable mais cela implique de gérer les communications entre les objets distants. Des outils tels que CORBA, RPC et Java RMI facilitent la mise en œuvre de la communication en masquant les accès réseaux. Cette maturation en termes de réseaux et de programmation par objets conduit aujourd'hui à une intensification du développement d'applications distribuées. Cette évolution des applications distribuées augmente le besoin de spécifier explicitement les sémantiques des communications et des interactions entre des objets.

Cependant, les outils mentionnés ci-dessus ne permettent pas d'exprimer les sémantiques des interactions entre des objets. Seuls quelques travaux vont dans le sens d'une expression et d'une gestion des interactions entre des objets distants indépendamment de leurs fonctionnalités intrinsèques. Cependant, il reste encore des travaux à faire sur la « sémantique » des interactions entre objets distants afin d'apporter encore plus de flexibilité, de facilité et une meilleure réutilisation lors de la mise en œuvre d'applications distribuées.

La solution avancée est la définition d'un modèle et d'une architecture distribuée de gestion des interactions entre objets distants dans les environnements de développement utilisés par le monde industriel, c'est-à-dire les environnements compilés, fortement typés et distribués. Elle est basée sur ISL (*Interaction Specification Language*), notre langage de description des interactions ainsi que sur un système de réécriture des comportements réactifs.

Mots-clefs : Architecture distribuée, Protocole à métaobjets, Environnements compilés et typés

Remerciements

Je tiens à remercier tout particulièrement les membres de mon jury pour l'intérêt qu'ils ont porté à mon travail. Je remercie également Paul FRANCHI-ZANNETTACCHI de m'avoir fait l'honneur de présider le jury.

Sincère remerciement à Anne-Marie PINNA-DÉRY pour m'avoir accepté dans son équipe et m'avoir conseillé et dirigé dans mes travaux de recherche tout au long de cette thèse. Que Mireille BLAY-FORNARINO reçoive ma profonde sympathie et reconnaissance pour son aide précieuse. Je leur dois, à toutes les deux, le goût de la programmation par interactions et de la recherche en général.

Je tiens à remercier chaleureusement Jean-Paul RIGAULT pour ses conseils avisés concernant la programmation à l'aide du langage C++ ainsi que Sabine MOISAN pour avoir cru au modèle MICADO et l'avoir utilisé dans l'une de ses applications.

Merci à l'ensemble des personnes qui m'ont accompagné sur ce chemin difficile qui conduit à la rédaction d'un mémoire de thèse. J'ai une pensée particulière pour Anne-Marie PINNA-DÉRY, Mireille BLAY-FORNARINO, Jean-Paul RIGAULT et Sabine MOISAN pour leur relecture, sans qui la qualité et la rigueur scientifique de ce manuscrit ne serait pas ce qu'elle est.

Je remercie chaleureusement ceux avec qui j'ai partagé des moments inoubliables : Annabelle, Christophe, Olivier, Pascal, Robert-Olivier, Xavier, et tous les autres ..., et qui, grâce à leur bonne humeur et leur amitié sincère, ont grandement contribué à l'ambiance formidable qui régnait, notamment lors de nos rencontres gastronomiques :-).

Bien évidemment, mille et un mercis à tous ceux qui ont partagé mon séjour à l'E.S.S.I. Je pense notamment à Xavier AUGROS et Pascal RAPICAULT pour nos (longues) discussions et leur témoignage d'amitié. Je remercie plus particulièrement Xavier AUGROS pour m'avoir supporté pendant les trois premières années de cette thèse.

Je remercie tout spécialement mes parents pour m'avoir activement apporté un soutien inconditionnel tout au long de mes études. Que cette thèse soit leur récompense tant méritée.

Je termine par une pensée particulière pour ma bien aimée. Je te remercie de tout mon cœur Isabelle pour avoir cru en moi durant les moments difficiles, pour m'avoir soutenu et épauler, pour ton sacrifice permanent lors de ces longs mois de rédaction.

À mon petit ange, à mes parents, ...

Laurent

Table des matières

Liste des figures et tableaux

Liste des figures

Liste des tableaux

Liste des définitions et propriétés

Liste des définitions

Liste des propriétés

INTRODUCTION GÉNÉRALE

Chapitre 1

Introduction générale

> « *Je vais vous annoncer une bonne nouvelle et une mauvaise nouvelle. La bonne nouvelle est que l'ordinateur est en voie de disparition. Mais ne vous réjouissez pas trop vite. L'ordinateur sera remplacé par quelque chose de bien pire : le réseau !*
> « *Je m'explique. La croissance exponentielle du nombre d'ordinateurs et de réseaux d'ordinateurs est un fait. En extrapolant cette tendance, on peut conclure que dans l'avenir l'ordinateur seul disparaîtra, pour être remplacé par un réseau de réseaux. [...] Un système réparti, c'est-à-dire un ensemble d'ordinateurs autonomes reliés par un réseau, est tout autre chose qu'un ordinateur tout seul.* » Peter Van Roy [Roy97].

Le but de cette thèse est de proposer un modèle permettant l'intégration des interactions dans un environnement distribué et basé sur des langages à objets compilés et fortement typés existants. De ce fait, dans ce chapitre, nous motivons tout d'abord l'intérêt des travaux présentés dans ce mémoire de thèse et en définissons clairement le contexte. Ensuite, nous posons le vocabulaire employé en donnant une définition précise au terme « *interaction* ». Nous poursuivons, de manière très succincte, par une présentation de notre objectif. Nous décrivons finalement, toujours de manière succincte, le plan de ce mémoire de thèse.

1.1 Problématique initiale

Depuis quelques années sont apparus des langages et des environnements de programmation pour construire des applications distribuées [1] (les *middleware*). Parmi ces langages et environnements, on peut citer Java [GJS96] (avec Java RMI), *.Net* framework [MS00] ainsi que le couple C++ [Str91] / CORBA [OMG98, GGM99].

Or, si on fait la synthèse de la première gamme d'outils, on se rend compte que ces outils intègrent les concepts de la programmation par objets, ceux de la programmation concurrente ainsi que ceux de la programmation distribuée. En effet, les notions d'abstraction, de modularité et de communication distante requises par les applications distribuées peuvent être assurées par les mécanismes fondamentaux des technologies objets que sont l'héritage, l'encapsulation et l'envoi de messages. Par exemple, en Java RMI [Dow98], la communication avec un objet distant est réalisée par l'envoi d'un message.

En fait, si ce type d'outils trouve actuellement un réel engouement c'est que ces outils sont une réponse aux nouveaux besoins des développeurs liés à la maturation en termes de réseaux et de programmation par objets. Cependant, les premières utilisations poussées de ces outils ont mis en évidence la difficulté, toujours présente, de réutiliser le code déjà existant.

Ainsi, bien que des outils tels que CORBA ou Java RMI facilitent la mise en œuvre de la communication en masquant certains détails des accès réseaux, certaines situations impliquent d'exprimer la coordination des activités (sémantique de la communication) qui dispose, au sein de ces outils, d'un support très restreint pour être spécifiée et d'un manque d'abstraction.

1. Dans ce mémoire de thèse, les termes *réparti* et *distribué* seront indifféremment utilisés.

Aussi, la programmation par composants tente d'améliorer de façon significative la réutilisation du code existant. En effet, elle vise à se rapprocher de la programmation « LEGO ® » (assemblage standardisé de composants) en définissant la notion de services. Ceci facilite indiscutablement la réutilisation des composants.

Cependant, et malgré cette avancée indéniable, nous arguons que l'une des faiblesses majeures est, actuellement, la non reconnaissance de la sémantique de la communication. Or, la prise en compte de cette sémantique nous paraît indispensable pour développer de réelles applications adaptables.

1.1.1 Portabilité, répartition et interopérabilité

Les nouvelles applications deviennent de plus en plus complexes. En effet, les enjeux actuels et futurs sont la gestion de systèmes distribués ouverts pouvant être configurés dynamiquement et capable de s'adapter par eux-mêmes à leur environnement d'exécution. En fait, il ne s'agit plus de développer une application monolithique mais, au contraire, de définir toute une architecture pour le développement d'un ensemble d'applications communicantes. Ceci implique notamment une mobilité accrue des composants constituant les applications (ils doivent pouvoir se déplacer d'un site à un autre) et une découverte dynamique de nouveaux services.

Par conséquent, le déploiement d'applications distribuées implique d'être capable d'ajouter ou de supprimer dynamiquement des relations entre différents objets (c'est-à-dire d'être capable de modifier dynamiquement la sémantique des communications entre objets), d'ajouter de nouveaux outils sans avoir à recompiler l'ensemble du système, de gérer des problèmes complexes de résistance aux pannes, de sécurité, etc., le tout dans un environnement hétérogène aussi bien en termes de plates-formes que de langages applicatifs.

Bien que la programmation orientée objet ait déjà prouvé ses intérêts lors de la mise en œuvre d'applications complexes, elle montre ses limites lors du développement de ces nouvelles applications. En réalité, la plupart des concepts non objets (tels que la notion de tâches d'exécution, de communication synchrone ou asynchrone) se retrouvent « éparpillés » dans le code de l'application. Par conséquent, la relecture et la maintenance du code en est d'autant plus difficile.

De plus, la mise en œuvre de ces concepts n'est pas portable d'une architecture à une autre (par exemple de CORBA à Java RMI). En fait, ils sont décrits au sein de ces architectures par le biais de mécanismes ad-hoc. Ainsi, les principes de la mise en œuvre de la sémantique des communications pour une architecture donnée ne peuvent être réutilisés tels quels par une autre architecture distribuée. Par conséquent, cette absence de portabilité et d'interopérabilité limite fortement l'extension et la maintenance des applications utilisant ces mécanismes.

1.1.2 Abstraction de la sémantique des communications

Lors de la phase de conception d'une application, la définition de la sémantique des communications joue un rôle important dans la modélisation des applications [BJR97] et est réalisée à un niveau élevé d'abstraction (par exemple sous la forme de scénarios). Lors de la phase de mise en œuvre, les langages à objets n'offrent pas, ou peu, de moyens pour exprimer cette sémantique à un niveau d'abstraction aussi élevé que celui proposé lors de la phase de conception.

En fait, cette sémantique est décrite, lors de la phase de mise en œuvre, dans le code des objets à l'aide de concepts objets tels que l'héritage, la délégation, la composition, etc. Elle perd ainsi sa notion d'entité conceptuelle propre et se trouve éparpillée au milieu du code décrivant les fonctionnalités intrinsèques des objets. Ceci provient du fait que la sémantique des communications est une propriété qui affecte un système dans son ensemble de manière transverse et orthogonale à la décomposition fonctionnelle (procédures et classes notamment) offerte par les langages de programmation.

Par conséquent, cette sémantique des communications se trouve donc mêlée dans le code des objets. Pour le programmeur, ceci complique de manière significative la maintenance, l'évolution et la réutilisation du code [DGV+95]. Elle n'est donc pas appréhendée par le programmeur de la même manière qu'un objet ou un module car elle ne dispose pas, au sein des langages à objets, d'une représentation au même niveau que les classes [KLM+97].

Or, l'inclusion, au sein d'un objet, de code spécifique pour décrire la communication entre cet objet et les autres rend le code de l'objet plus difficile à comprendre. De plus, le code de l'objet est rendu plus complexe à chaque nouvelle interaction [DGV+95]. Ce très fort couplage entre fonctionnalités

intrinsèques des objets et communications inter objets inhibe fortement les concepts de base des modèles de classes que sont la réutilisation, l'évolution et l'encapsulation.

Pour pallier les limites des technologies à objets, des outils tels que CORBA, Java RMI, DCOM [Bro94] ou les EJB [MH99], proposent une abstraction de la communication dans des environnements distribués sous la forme de services. Cependant, cette abstraction est loin d'être suffisante. En effet, nous pouvons observer, en analysant les programmes écrits dans un langage orienté objet, qu'une fraction significative du code des méthodes est dévolue à la communication de l'objet avec les autres objets, rendant ceux-ci interdépendants.

1.1.3 Réalités industrielles

Des travaux autour d'une expression de la sémantique des communications indépendamment des objets ont déjà été menés. Cependant, comme nous le verrons dans la partie I, ceux-ci, issus de la recherche, ne prennent pas, ou peu, en compte les besoins du monde industriel (environnements distribués et hétérogènes notamment). La principale raison à ce « décalage » provient du fait que ces travaux se restreignent volontairement à des environnements beaucoup moins complexes afin de se focaliser sur les propriétés essentielles qui doivent être vérifiées pour permettre d'exprimer simplement la sémantique des communications. Cette première étape est bien entendu indispensable.

Cependant, grâce à ces travaux, il nous semble désormais possible [2] de passer à l'étape suivante, celle qui permet d'intégrer l'expression de la sémantique des communications dans un environnement de programmation industriel. Or, actuellement, l'un des principaux leitmotivs du monde industriel concerne la réutilisation et l'assemblage de composants logiciels indépendants, et ce dans un environnement distribué et hétérogène.

Par conséquent, de nombreux travaux sont à mener dans cette direction pour que les concepts novateurs des travaux ci-dessus deviennent une sérieuse alternative à des environnements comme CORBA ou Java / Java RMI. Il est à noter qu'il ne s'agit pas de réinventer ces outils mais, au contraire, de les encapsuler à l'aide d'une couche offrant le niveau d'abstraction de la sémantique des communications souhaité.

Ainsi, nos travaux, présentés dans ce mémoire de thèse, ont pour principal objectif de proposer une intégration de l'expression de la sémantique des communications dans un environnement proche des réalités industrielles. Nous nous focalisons donc sur des environnements compilés, fortement typés et distribués basés sur C++ / CORBA ou Java / Java RMI en mettant notamment l'accent sur le support de l'interopérabilité des systèmes.

Enfin, l'une des autres préoccupations majeures du monde industriel concerne les performances des applications. En nous positionnant volontairement dans le giron du monde industriel, nous nous devons d'offrir une solution dont l'empreinte globale est la plus faible possible sur l'application, et ce aussi bien lors de la phase de conception (afin de permettre une réutilisation maximale du code déjà existant), que lors de l'exécution (afin de ne pas dégrader de manière significative les performances).

1.2 Sémantique des communications : un vocabulaire riche

La littérature utilise indifféremment les termes *relations* [Rum87], *associations* [BEL+95, Tan95], *dépendances* [Duc97b, DR97, LB97] ou *interactions* [FP90, JSH+96] pour désigner la sémantique des relations entre objets bien que chaque terme ne revêt pas nécessairement la même signification. Dans ce paragraphe, nous spécifions et motivons notre terminologie.

À la différence du concept d'interaction qui désigne un comportement dynamique, les notions de relation et d'association revêtent plutôt un caractère statique (même si les relations peuvent évoluer dans le temps) et sous-entend une notion d'ordre (relations orientées) : « *a relation as any connection from one object `Or` to another objet `Os` that influences the behaviour of `Or` in some way.* » [Bos95c].

Les dépendances, quant à elles, ne sont, en réalité, qu'un cas particulier de relations. Elles décrivent en fait des relations n-aires par le biais d'une entité : « *une dépendance est une entité responsable de l'interconnexion d'un ensemble d'objets, entité qui influence le comportement de ces objets.* » [Duc97b]. On peut noter que la notion d'ordre induite par la définition d'une relation est toujours présente dans la définition des dépendances, même si elle est plus nuancée.

2. Et le but de ce mémoire de thèse est de vous le prouver...

À notre avis, cette notion d'ordre ne doit pas être présente. En effet, il nous semble que la relation ou la dépendance doit permettre aux objets sur lesquels elle s'applique de s'influencer mutuellement, au lieu que ce soit elle qui les influence.

Ainsi, de par sa définition, « *action, influence réciproque d'une chose sur une autre.* » (Pluridictionnaire Larousse, édition 1985), nous utilisons le terme « *interaction* » pour parler de relations ou de dépendances entre objets. Nous avons choisi ce terme car il montre parfaitement que les objets interagissants s'influencent mutuellement, spécifiant clairement que le comportement d'un objet interagissant est déterminé en fonction, non seulement de l'interaction, mais aussi des autres objets interagissants. Ainsi, nous définissons le terme interaction comme suit : « *une interaction est une entité ayant la charge d'interconnecter un ensemble d'objets afin que leurs comportements s'influencent mutuellement.* ».

1.3 Objectif

Le but de cette thèse est l'étude, la définition et la mise en œuvre du concept d'interactions dans les environnements de développement utilisés par le monde industriel, c'est-à-dire des environnements compilés, fortement typés et distribués tels que C++ [Str91] ou Java [GJS96] pour les langages de programmation et CORBA [OMG98] ou Java RMI [Dow98] pour les architectures distribuées.

Cette réflexion autour du concept d'interactions nous a amené à définir un modèle dont le maître mot est la possibilité d'adapter facilement une application existante afin de permet l'ajout et la suppression dynamique d'interactions. Cependant, cette réflexion se veut suffisamment générique afin que le modèle puisse être facilement projeté dans d'autres contextes (caractéristique d'autant plus essentielle que notre cadre d'étude se situe dans un monde très actif, en perpétuel évolution).

Ainsi, ce mémoire de thèse se compose de 4 parties. Nous les présentons ci-dessous succinctement. Le lecteur désireux d'obtenir de plus amples informations pourra se reporter à la page « Dans cette partie vous trouverez... » de chacune de ces parties. La première partie présente une analyse de l'existant sur le support des interactions dans les modèles à objets, à composants ou distribués. De cette analyse est basé notre modèle à interactions distribuées, présenté, de manière formelle, dans la seconde partie. La troisième partie décrit une mise en œuvre possible, en C++ et CORBA, du modèle à interactions distribuées et des mécanismes qu'il nécessite. La quatrième, et dernière, partie présente un ensemble d'applications utilisant des interactions.

1.3.1 Une étude de l'existant

Certains travaux proposent une meilleure prise en compte des interactions dans les modèles à objets, à composants ou à agents et dans des environnements « séquentiels », concurrentiels ou distribués. Ces travaux gravitent autour de deux axes principaux : la définition de nouveaux langages de programmation et l'utilisation de techniques dérivées des concepts de la programmation orientée objet. Ces travaux offrant un support aux interactions sont analysés au **chapitre 2**. De cette analyse est extrait un ensemble de critères qu'il nous semble important de vérifier afin d'offrir le meilleur support possible aux interactions.

Le **chapitre 3** situe notre environnement de travail (C++ / Java avec CORBA / RMI) vis-à-vis des critères ci-dessus. Il présente également différentes mises en œuvres possibles du support des interactions à l'aide des divers mécanismes de CORBA et en déduit, à chaque fois, les carences et les limites. Il propose, en guise de conclusion, une introduction au modèle à interactions distribuées que nous définissons dans la partie suivante.

1.3.2 Un modèle à interactions distribuées

L'objectif de cette partie est la définition d'une part, d'un modèle à interactions distribuées offrant une sémantique claire aux interactions et ce dans un contexte proche des réalités industrielles tel que C++ avec CORBA, mais aussi Java et Java RMI et, d'autre part, d'un langage dédié à la description des interactions : *Interaction Specification Language* (ISL).

Avant d'entrer dans le vif du sujet avec la description formelle du modèle, le **chapitre 4** donne une idée concrète de l'utilisation de notre modèle dans le monde compilé et fortement typé qu'est C++ et CORBA.

Ensuite le **chapitre 5** propose une définition formelle de notre modèle à interactions distribuées. Celle-ci est complétée par le **chapitre 6** qui décrit la fonction de fusion comportementale des règles

d'interaction. Il présente en particulier les règles de réécriture (en sémantique naturelle) qui décrivent la sémantique de la fusion comportementale des règles d'interaction.

1.3.3 Une architecture de mise en œuvre

Cette partie décrit une mise en œuvre de l'architecture et du protocole que nous avons proposé pour la gestion des interactions. Ainsi nous montrons comment les interactions s'intègrent dans le modèle à objets et dans une architecture distribuée.

Le **chapitre 7** présente les deux principales techniques utilisées par notre architecture pour mettre en œuvre les interactions : la réflexivité et la programmation par aspects – *Aspect Oriented Programming* (AOP) [KLM+97].

Le **chapitre 8** présente la manière dont les interactions sont représentées dans notre architecture. De cette représentation, nous pouvons définir un ensemble de classes (et métaclasses) définissant les fondations de notre architecture (noyau minimal).

Le **chapitre 9** se concentre sur la mise en œuvre de la partie de l'architecture du modèle à interactions distribuée concernant la gestion de l'exécution des interactions dans l'environnent C++ / CORBA. Il complète donc la description de l'architecture du modèle à interactions distribuées définie dans la partie précédente.

1.3.4 Des applications

Cette quatrième et dernière partie présente des utilisations possibles des interactions au travers d'une application de génération d'interfaces graphiques pour des systèmes à base de connaissances distribués.

Le **chapitre 10** présente une application développée, à l'aide des interactions, dans le cadre du projet COLOR. Les interactions ont été utilisées pour interfacer un système à base de connaissances (développé en C++) avec une application de visualisation des résultats (développé en Java). Le bus logiciel CORBA est utilisé comme couche réseau.

1.3.5 Un bilan, des perspectives

Ce mémoire de thèse se termine (**chapitre 11**) par un résumé de la contribution apportée (bilan général) et par une description des travaux futurs.

Des annexes

Ce document comporte trois annexes. L'**annexe A** présente les interfaces IDL [Lam87] complètes des opérateurs réactifs (présentés dans le chapitre 5), l'**annexe B** décrit le protocole d'enregistrement des règles d'interactions auprès d'un métaobjet, tandis que l'**annexe C** propose une solution dans CORBA pour rendre accessible à distance un objet non prévu pour l'être initialement.

Partie I

ÉTUDE DE L'EXISTANT : SUPPORT DES INTERACTIONS

Dans cette partie vous trouverez ...

Le développement d'applications distribuées à l'aide des technologies objets est rendu difficile en raison de l'hétérogénéité des plates-formes et des langages et de la complexité des environnements répartis. La maîtrise de la réutilisation et de l'interopérabilité des objets est un enjeu majeur pour des mécanismes totalement adaptés à la répartition des objets et composants et à leur communication.

Ce type d'applications implique d'exprimer la coordination des activités. Or celle-ci dispose d'un support très restreint pour être spécifiée et manque de puissance d'expression de l'abstraction. En fait, les modèles de programmation mettent généralement l'accent sur les concepts d'objets, de composants ou d'agents et délaissent, la plupart du temps, les interactions [Sha94].

Cependant, actuellement, la notion d'interaction n'a pas d'abstraction dans les concepts objets et ne dispose pas d'une représentation au même niveau que les classes. Elle se trouve mêlée dans le code des objets interagissants ce qui, pour le programmeur, complique de manière significative la maintenance, l'évolution et la réutilisation du code.

De plus, la complexité du code de l'objet est augmentée à chaque nouvelle interaction. Ce très fort couplage entre fonctionnalités intrinsèques des objets et interactions inhibe fortement les concepts de base des modèles de classes que sont la réutilisation, l'évolution et l'encapsulation.

Certains travaux proposent une meilleure prise en compte des interactions dans les modèles à objets, à composants ou à agents et dans des environnements « séquentiels », concurrentiels ou distribués. Ces travaux gravitent autour de deux axes principaux : la définition de nouveaux langages de programmation et l'utilisation de techniques dérivées des concepts de la programmation orientée objet. Ces travaux offrant un support aux interactions sont analysés au **chapitre 2**. De cette analyse sont extraits des critères qu'il nous semble important de vérifier afin d'offrir le meilleur support possible aux interactions.

Le **chapitre 3** situe notre environnement de travail (C++ / Java avec CORBA / RMI) vis-à-vis des critères ci-dessus. Il présente également différentes mises en œuvre possibles du support des interactions à l'aide des divers mécanismes de CORBA et en déduit, à chaque fois, les carences et les limites. Il propose, en guise de conclusion, une introduction au modèle à interactions distribuées que nous définissons dans la partie II.

Chapitre 2

Étude comparative des approches offrant un support aux interactions

> « *Whereas the implementation relationship is concerned with how a component achieves its computation, the interaction relationship is used to understand how that computation is combined with others in the overall system.* » [AG94]

> « *Component software is becoming an increasingly popular choice for system development, its goal being the development of highly reusable, customizable software components.* » [SML99]

Ce chapitre présente une étude de différentes approches offrant un support aux interactions. Bien que notre travail se focalise sur les langages à classes dans un environnement compilé, fortement typé et distribué, notre étude ne se restreint pas à ce domaine mais effectue une analyse couvrant un large spectre allant des technologies objets aux concepts d'agents, des environnements interprétés à ceux compilés, des systèmes « séquentiels » aux systèmes distribués. Le but étant de déterminer les critères qui devront être vérifiés par notre modèle afin d'offrir un support « idéal » aux interactions. Une partie de cet état de l'art a été soumise sous la forme d'un article dans la revue l'Objet [Ber01].

Dans la suite, les critères seront notés sous la forme Ci.j dès leur apparition. Chacune des synthèses partielles présentes dans ce chapitre indique, dans un tableau, si les approches les plus représentatives sur lesquelles elle porte supportent (symbole ✓), ne supportent pas (symbole ✗) ou supportent sous certaines conditions (symbole ✘) chacun des critères obtenus de l'analyse de l'ensemble des approches présentées dans ce chapitre. En effet, ceux sont ces critères qui nous permettent de déterminer l'adéquation d'une approche vis-à-vis d'un support « idéal » des interactions.

2.1 Support des interactions dans les langages à objets

Il est maintenant parfaitement accepté que les technologies orientées objet améliorent sensiblement la réutilisation et l'extension du code [BR89, RBP+91] par rapport aux technologies procédurales ou fonctionnelles. Ces améliorations sont réalisées grâce à l'encapsulation et au polymorphisme. En outre, elles offrent une abstraction des principaux concepts mis en œuvre [Car95]. Cependant, avec la complexité toujours grandissante des systèmes actuels, les propriétés offertes par les technologies objets montrent leurs limites, notamment au sujet des interactions.

2.1.1 Expression des interactions dans le modèle à objets

Les problèmes inhérents au paradigme objet, tels que l'anomalie de l'héritage [Sny86, MY93, McH94] dans les langages objets concurrents, ou les aspects transversaux [KLM+97], limitent la réutilisation des objets. De plus, des difficultés apparaissent lorsque des objets hétérogènes (c'est-à-dire d'origines diverses) conçus pour différents contextes applicatifs doivent être intégrés ensemble, en particulier si ces objets sont dépendants de leur contexte d'exécution.

Certains de ces problèmes proviennent du fait que les interactions (c'est-à-dire les relations qui existent entre les objets) sont insérées dans les différentes parties de code fonctionnel et donc cachées (et par conséquent difficilement identifiables), dans le code des objets. Ce mélange, dans les approches objets traditionnelles, des interactions dans le code des objets provient du fait que les interactions, en tant que schéma de programmation, ne s'ajustent ni avec l'approche par objet, ni avec l'approche procédurale sous-jacente aux langages à objets. En fait, les interactions ne sont pas des éléments de décomposition fonctionnelle du système mais des propriétés qui affectent la sémantique des objets d'une manière systématique.

De plus, alors que la description des interactions dispose d'une identité logique et conceptuelle lors de la phase de conception (par exemple grâce aux scénarii), celle-ci est absente lors de la mise en œuvre des applications à l'aide des langages orientés objet. Le paradigme objet n'offre, en réalité, qu'un support très restreint à la représentation de l'architecture logicielle d'une application. Seul le concept de hiérarchie des classes reste présent lors de la mise en œuvre à l'aide d'un langage de programmation orienté objet.

En vérité le modèle à objets « conventionnel » n'offre aucun réel support aux interactions. Le code servant à décrire la sémantique des interactions est « saupoudré » dans celui des objets interagissants. Ceci implique que le code d'un objet contient non seulement le code décrivant ses comportements intrinsèques, mais aussi le code décrivant la sémantique des interactions auxquelles l'objet participe. Ainsi le contrôle des interactions, contenues dans le code des objets, est fortement limité et rend les objets dépendants des autres objets interagissants.

2.1.2 Expression des interactions dans l'interface des objets

Les approches Composition Filters [Ber94, ABV92, AT98, AB98], PROCOL [BL91], Software Adaptors [YS94], Coordinating Environments [Muk95], Layered Object Model (aussi appelé LAYOM) [Bos95a, Bos95b] et Jam [ALZ00] décrivent les interactions dans l'interface des objets. Les trois premières approches les décrivent sous la forme de règles déclaratives (des filtres pour les deux premières, des protocoles pour PROCOL). Ces règles sont définies dans l'interface des objets interagissants et donc directement associées aux objets auxquels elles s'appliquent. Jam, quant à elle, décrit une interaction sous la forme d'un *mixin*, c'est-à-dire une interface Java étendue (Jam est une approche basée sur Java [CH96]).

Avec les Composition Filters, par exemple, les filtres d'entrée d'un objet permettent de spécifier quelle est la méthode de l'objet filtré ou des objets internes ou externes (se reporter au langage Sina [TBA89, Koo95] pour une description complète du modèle de filtres des Composition Filters) qu'il faut exécuter lors de la réception d'un message entrant. Ainsi il est possible, pour chaque message reconnu par l'objet filtré, de définir un filtre qui redirige le message entrant vers la méthode contenant la description de l'interaction associée à ce message. Ceci permet de disposer de toute la puissance d'expression du langage pour décrire l'interaction.

Le fait de décrire les interactions dans la partie interface de l'objet permet de les séparer du code fonctionnel de l'objet (et offre ainsi une meilleure abstraction). Cela permet le **support de l'héritage des comportements des interactions** (issu de l'héritage des comportements définis dans les filtres ou les protocoles). Ainsi, les interactions peuvent être définies de manière incrémentale, mais uniquement par extension des comportements qu'elles décrivent. Il est à noter, cependant, que les *mixins* de Jam ne peuvent pas directement hériter des comportements d'autres *mixins*. En réalité, un *mixin* ne peut hériter que d'une classe Java. Ainsi, l'héritage des comportements d'une interaction par un *mixin* implique l'utilisation d'une classe intermédiaire.

Pour toutes ces approches, la sémantique d'une interaction est décrite par une méthode d'un objet, tandis que l'interaction elle-même est décrite grâce à la définition d'une règle sur l'objet receveur. Avec cette solution **une interaction est fortement liée aux objets interagissants** sur lesquels elle agit puisque l'interface de ces objets doit être modifiée pour y définir les règles nécessaires à l'interaction.

Critère C1.1 : Couplage objets / interactions

Le couplage objets / interactions doit être le plus faible possible. Ceci doit se traduire notamment par une définition et une déclaration des interactions indépendantes de celles des objets sur lesquels elles s'appliquent.

Ainsi, **les interactions sont définies en même temps que les objets** sur lesquels elles s'appliquent. De ce fait, elles ont une durée de vie identique à ces objets. Ceci implique que, lorsque l'interaction ne doit être présente (active) qu'à certains moments de l'exécution (par exemple lorsqu'une condition est vérifiée), c'est la règle, et par conséquent l'objet filtré, qui a en charge de déterminer si l'interaction est active ou non. Ainsi, la sémantique de l'interaction se trouve encore en partie incluse dans celle des objets interagissants. Cette dernière contrainte ne permet pas une réutilisation maximale des objets car ils sont liés à d'autres objets par le biais des interactions qui leur sont associées.

EXEMPLE. — L'exemple de la figure 2.1 déclare un objet représentant un compte bancaire (objet de classe `AccountItf`) et un objet représentant la sémantique d'un virement bancaire (objet de classe `BankOffice`). L'interaction se trouve dans l'interface de ce second objet et se compose de deux filtres d'entrée, l'un filtrant les opérations concernant l'objet filtré (filtre `disp`), l'autre filtrant les crédits et débits (filtre `trans`). L'objet de virement bancaire dispose d'une référence sur le compte bancaire sur lequel s'effectue le virement ainsi que d'un objet interne, `theManager`, contenant la sémantique du transfert (par exemple sous la forme d'une opération atomique).

Le filtre `disp` exécute les méthodes internes de l'objet (nommé `inner`) en utilisant la sémantique de communication `Dispatch` (appel de méthodes). Le filtre `trans` décrit le comportement de l'objet lors de l'appel d'une méthode sur l'objet (sémantique de communication `Meta`). La condition d'activation du filtre est `True` ce qui signifie que le filtre (et donc l'interaction) est toujours actif. Le comportement décrit dans le filtre indique que, lors de l'appel de la méthode `Transfer` sur l'objet `theAccount`, la méthode `moneyTransfer` de l'objet filtré (`inner`) sera appelée en lieu et place de la méthode initiale, les paramètres étant réifiés sous forme d'un `Message`. C'est cette méthode qui va contenir la sémantique de l'interaction.

```
class AccountItf interface
  methods
    Deposit (Float) returns Nil;
    Withdrawal (Float) returns Nil;
  inputfilters
    disp : Dispatch =
              { True => inner.* };
end;

class BankOffice interface
  externals
    theAccount : AccountItf;
  internals
    theManager : TransferAgentItf;
  methods
    moneyTransfer (Message)
      return Nil;
  inputfilters
  {
    disp : Dispatch = { True => inner.* };
    trans : Meta = { True => [theAccount.Transfer]inner.moneyTransfer };
  }
end;
```

```
class AccountItf implementation
  instvars
    AccountNumber : Integer;
  methods
    Deposit (amount : Float)
      begin ... end;
    Withdrawal (amount : Float)
      begin ... end;
end;

class BankOffice implementation
  methods
    moneyTransfer (m : Message)
      begin
        ...
      end;
end;
```

FIG. 2.1 – *Exemple d'interaction avec le modèle des* Composition-Filters

Pour résumer, avec ces quatre approches, les interactions sont définies et déclarées en même temps que les objets sur lesquels elles s'appliquent (dans leur interface). Ceci implique encore un assez fort couplage objets / interactions et limite ainsi l'abstraction des interactions et l'indépendance des objets vis-à-vis de celles-ci. Une conséquence de ceci est que la durée de vie d'une interaction est identique à celle de l'objet définissant l'interaction au sein de son interface. Une autre conséquence est l'impossibilité d'ajouter dynamiquement des interactions en cours d'exécution car aucun mécanisme n'est défini pour modifier dynamiquement l'interface d'un objet et, plus particulièrement, ses règles. En contrepartie, **l'exécution des interactions est entièrement gérée par le système**, et donc l'objet émetteur d'un message n'a pas à se soucier de celles-ci.

Critère C2.1 : Transparence de l'exécution des interactions

Le support des interactions doit être transparent. Ceci doit se traduire par la totale prise en charge du support nécessaire à l'exécution des interactions, et ce sans que le programmeur soit obligé de déclencher explicitement une interaction.

Ainsi, avec ces approches les interactions disposent d'un niveau insuffisant d'abstraction (mais supérieur à celui offert par le modèle à objets). En effet, elles restent encore fortement liées aux objets sur lesquels elles s'appliquent avec les conséquences que cela implique. D'autres approches, décrites ci-après, ont poussé plus loin l'abstraction des interactions en les définissant dans une entité spéciale et indépendante des objets interagissants.

2.1.3 Expression des interactions par extension du modèle à objets

Les approches COM [Bro94], Java Beans [Eng97, SUN97], CORBA Component Model [1] (CCM) [Mar99, OMG99] ainsi que le modèle à composants [EHT95, SML99] étendent le modèle à objets en lui ajoutant une spécification très précise des moyens de communication entre objets : les interfaces pour COM, les flots d'événements (*event streams*) pour Java Beans, les facettes pour CCM, les entités de liaisons [MBVD+97, GHM+99] (généralement nommées *connecteurs*) pour le modèle à composants. Dans les quatre cas, ceux-ci sont plus ou moins associés à un objet. Par conséquent, ceci influence de manière significative les relations qui existent entre les interactions et les objets interagissants.

Ainsi, dans le cas de COM, les interactions étant définies dans les interfaces des composants, nous avons un très **fort couplage objets / interactions**. Nous retrouvons cette approche avec la notion de *ports* de CCM. Cependant les composants de CCM disposent également, comme moyen de communication, des *canaux d'événements* de CORBA (Notification Service [OMG97b]). Ceci offre un début d'abstraction des interactions. Les architectures Java Beans et CCM permettent, quant à elles, de dissocier la définition des interactions de celles des composants.

Dans le modèle à composants, les connecteurs sont les entités complémentaires aux composants [AG94, Sha95] permettant leur assemblage en vue de leur interconnexion. Ils sont généralement considérés comme la « colle » liant les composants interagissants [KCA+96, MG96]. Alors que les composants sont les unités de traitement des données, les connecteurs contiennent la sémantique de la communication permettant à ces composants de travailler ensemble.

Les connecteurs du modèle à composants ou les *beans* de Java Beans correspondent donc à des **entités logicielles mettant en œuvre les relations entre composants**. Ils peuvent ainsi contrôler le respect par la communication de certaines règles préétablies par le système afin d'assurer la cohérence des communications. Ils permettent notamment de vérifier la compatibilité, au sens du typage, des paramètres en sortie du composant appelant avec ceux en entrée du composant appelé, d'adapter ces paramètres (par transtypage notamment) afin que la connexion puisse être définie entre des composants, ou de mettre en œuvre différents protocoles de communication.

EXEMPLE. — La figure 2.2 présente un composant OLE offrant plusieurs interfaces (trois sont présentées). L'obtention d'une interface est réalisée par une invocation de la méthode `QueryInterface` définie dans chaque interface d'un composant OLE. L'identifiant de l'interface est fournie comme paramètre à cette méthode. Ces interfaces font partie intégrante du support des interactions par OLE.

Il est à noter que certaines interfaces peuvent permettre la création d'autres interfaces du composant. Ainsi, les interactions peuvent être dynamiquement créées, mais sont statiquement définies.

```
interface CShellContextMenu :
  IContextMenu
{
  public:
    STDMETHODIMP QueryContextMenu
      (HMENU, UINT, UINT, UINT, UINT);
    STDMETHODIMP InvokeCommand
      (LPCMINVOKECOMMANDINFO);
    STDMETHODIMP GetCommandString
      (UINT, UINT, UINT, LPSTR, UINT);
    STDMETHODIMP GetCommandString
      (UINT, UINT, UINT *, LPSTR,
       UINT);
};
```

```
interface CShellPropSheetExt :
  IShellPropSheetExt
{
  public:
    STDMETHODIMP AddPages ( ... );
    STDMETHODIMP ReplacePage ( ... );
};

interface CShellExtInit : IShellExtInit
{
  public:
    STDMETHODIMP QueryInterface
      (REFIID, LPVOID FAR*);
};
```

```
STDMETHODIMP CShellExtInit::QueryInterface (REFIID riid, LPVOID FAR *ppvObj)
{
  if (riid == IID_IShellExtInit)
    *ppvObj = this;
  else if (riid == IID_IShellPropSheetExt)
  {
    CShellPropSheetExt *ShellPropSheetExt = new CShellPropSheetExt ();

    hRes = ShellPropSheetExt->QueryInterface (riid, ppvObj);
  }
  else if (riid == IID_IShellContextMenu)
    ...
}
```

FIG. 2.2 – *Exemple d'interaction avec OLE*

Dans l'approche Java Beans, chaque *bean* peut supporter un ou plusieurs flots d'événements, auxquels d'autres beans peuvent souscrire. Ce support de flots d'événements est facultatif dans CCM (qui propose les notions de ports et de facettes comme moyen de communication). Ces flots d'événements permettent à un bean d'envoyer un événement (un message) via un flot d'événements, aux différents beans qui sont connectés à ce flot d'événements. Un flot d'événements est donc un connecteur. L'abon-

1. Le modèle CCM est basé sur le modèle des *Enterprise Java Beans* (EJB) [MH99].

nement (ou le désabonnement) d'un bean à un flot d'événements est une opération dynamique. Ceci est réalisé par le bean générateur d'événements dans le flot d'événements. Ainsi il est possible de définir un bean dont le rôle va consister à émettre des événements sur des flots d'événements à partir d'événements générés par d'autres beans. Ce bean peut donc être vu comme une interaction.

Cette définition des interactions sous forme de *beans* dans Java Beans permet de définir dynamiquement de nouvelles interactions grâce au chargement dynamique de classes du langage Java [CH96]. Ceci n'est pas possible avec COM puisqu'un objet COM dispose d'un ensemble d'interfaces, chaque interface décrivant la sémantique d'une interaction (et jouant par conséquent le rôle des connecteurs du modèle à composants).

Il est à noter que **la déclaration dynamique d'une interaction** avec l'architecture des Java Beans ou de CCM **est contrainte par la liste des événements générés par un composant**. En effet, une interaction dont le rôle est de réagir à une action $\mathcal{A}$ déclenchée sur un objet implique que cet objet génère un événement lorsque l'action $\mathcal{A}$ est déclenchée afin qu'il puisse être intercepté par l'interaction. Si un tel événement n'est pas généré, l'interaction ne pourra pas être ajoutée dynamiquement au système. Ainsi, ces modèles ne proposent pas de connecteurs explicites mais juste des connexions entre des ports ou des canaux événementiels compatibles.

Critère C2.1' : Préparation des objets pour interagir
Il ne doit pas être nécessaire de préparer explicitement (c'est-à-dire par le programmeur) les objets afin qu'ils puissent devenir interagissants. Ceci doit se traduire par la possibilité théorique de déclarer n'importe quelle interaction sur n'importe quel objet.

L'obligation qui est faite, par l'architecture des Java Beans, aux beans de communiquer entre eux par le biais d'événements permet d'assurer l'exécution transparente des interactions. Cette propriété de transparence est aussi vérifiée en COM car les objets COM ne sont accessibles qu'au travers d'interfaces et que ce sont ces interfaces qui jouent le rôle d'interactions. Le modèle CCM offre les deux solutions ci-dessus pour communiquer. Par conséquent, il permet lui aussi l'exécution transparente des interactions.

2.1.4 Synthèse partielle

Le tableau 2.1 présente une synthèse du support des interactions par les approches étudiées dans ce paragraphe les plus représentatives.

Critères														
Abstraction des interactions				Intégration dans le modèle à objets				Expressions des interactions				Aspects dynamiques		
C1.1	C1.2	C1.3	C1.4	C2.1	C2.2	C2.3	C2.4	C3.1	C3.2	C3.3	C3.4	C4.1	C4.2	C4.3
Modèle à objets [INR98]														
✗	✗	✗	✗	✗	✗	✗	-	✗	✗	✗	✗	✗	✗	✗
Modèle à composants [SML99]														
✓	✗	✗	✘	✓	✓	-	-	✗	✗	✗	✘	✘	✘	✗
Composition Filters [ABV92]														
✗	✗	✗	✗	✓	✗	✓	-	✓	✗	✓	✗	✗	✗	✗
Java Beans [Eng97]														
✓	✓	✗	✓	✗	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗

✓ : supporté, ✗ : non supporté, ✘ : supporté sous certaines conditions

TAB. 2.1 – *Synthèse du support des interactions dans les langages à objets*

2.2 Support des interactions par le biais d'une entité spécifique

Certaines approches offrent un support aux interactions sans nécessiter d'étendre la sémantique d'un langage à objets (ceci ne signifie pas que ces approches n'étendent pas la syntaxe du langage). Elles se basent sur l'utilisation, généralement combinée, des concepts définis par les modèles à objets, à composants, ou à agents.

2.2.1 Expression des interactions dans une entité distincte

Certaines approches décrivent les interactions dans une entité permettant d'abstraire les interactions. Parmi ces approches l'on peut citer RINs [SR92], Contracts [HHG90, Hol92], Synchronizers [FA93] et FLO [DBD95, Duc97b]. Chacune utilise un vocabulaire pour décrire l'interaction : un objet nommé *synchronizer* en Synchronizer, *dépendance* en FLO par exemple.

Ces approches permettent ainsi la dissociation des interactions vis-à-vis des objets interagissants et donc de rendre les **objets indépendant des interactions** auxquelles ils vont participer. Cette indépendance entre objets interagissants et interactions implique que la spécification des interactions ne peut porter que sur les interfaces des objets ce qui assure un respect du concept d'encapsulation du modèle à objets.

Critère C2.2 : Respect des paradigmes du modèle sous-jacent
Les interactions doivent respecter les paradigmes du modèle sous-jacent et en particulier l'encapsulation. Ceci doit se traduire par la possibilité de ne définir les interactions qu'en termes des interfaces des objets interagissants.

Comparées aux autres approches, FLO va encore plus loin dans l'abstraction des interactions en réifiant les entités les décrivant sous la forme d'objets de première classe (ce qui n'était pas le cas des autres approches). Ainsi les interactions peuvent être manipulées comme toutes les autres entités du système. Une interaction peut donc, par conséquent, être instanciée sur un ensemble d'objets interagissants lorsque la sémantique qu'elle définit doit être appliquée à ces objets, puis détruite lorsque sa sémantique n'est plus utilisée. Nous avons ainsi, dans ce cas, la possibilité de déclarer (instancier) les interactions indépendamment des objets interagissants.

Critère C1.1' : Interactions = entités de première classe
Les interactions doivent disposer d'un niveau élevé d'abstraction. Ceci doit se traduire par l'expression des interactions sous la forme d'une entité disposant de l'intégralité de la sémantique de la communication décrite par l'interaction.

NOTE. — La vérification de ce critère implique nécessairement la vérification des critères C2.1 et C2.2.

Avec ces approches, les comportements interagissants sont **décrits de manière déclarative**. FLO, par exemple, décrit les interactions sous la forme de règles réactives (en STklos [Gal96], langage objet basé sur CLOS [BDG+88, BKdR91] et Scheme [ADH+98]). Bien que ces approches n'offrent pas toute la puissance d'expression du langage utilisé pour décrire les interactions, elles proposent en échange des constructions sémantiques (des opérateurs réactifs pour FLO) qui leur sont propres et qui, au final, procurent au programmeur un environnement d'expression des interactions plus riche car spécialisé pour ce domaine [Duc97b]. Ceci implique une gestion par le langage à objets de l'exécution des interactions.

Alors que le concept d'héritage des comportements des interactions était implicite pour les approches exprimant les interactions dans l'interface des objets, il n'en n'est pas de même pour les approches décrites dans ce paragraphe. En fait seuls les *synchronizers* et les *dépendances* de FLO supportent la notion d'héritage des interactions. Les autres approches n'offrent pas de réels moyens pour qu'une interaction puisse hériter des comportements d'autres interactions. Un exemple est l'approche RINs qui, elle non plus, ne gère pas cet héritage et la réutilisation des spécifications des interactions. Elle décrit des rôles, en termes de méthodes disponibles, et des relations, en termes d'objets « partenaires ». Les interactions entre les rôles sont basées sur le modèle Petri-Net [Pet77] qui ne possède pas de notion d'héritage entre des graphes Petri-Net (un graphe décrivant une interaction).

Critère C1.2 : Héritage des interactions
La description des interactions doit pouvoir être réalisée de manière incrémentale, par raffinement, spécialisation et extension. Ceci doit se traduire par l'adaptation aux interactions du concept d'héritage des objets défini par le modèle à objets.

EXEMPLE. — L'exemple de la figure 2.3 (issu de [Duc97b]) montre une interaction en FLO décrivant le comportement d'exclusion mutuelle entre un groupe de boutons poussoirs afin qu'il n'y ait toujours au plus qu'un seul bouton sélectionné. La dépendance (définie par le mot-clef `deflink`) de cet exemple spécifie qu'un bouton n'est sélectionné que si aucun autre bouton ne l'est déjà.

Pour ce faire l'exécution du message permettant de sélectionner un bouton n'est possible (opérateur `permitted-if`) que si aucun autre bouton du groupe de boutons géré par la dépendance n'est sélectionné. L'état qui représente le fait que l'un des boutons de la dépendance est, ou non, sélectionné est mémorisé dans la variable `active?`. Elle est mise à jour chaque fois qu'un bouton reçoit un message (opérateur `implies`). La méthode `action-before-effective` permet de s'assurer que les boutons participant à une dépendance sont dans un état cohérent lorsque l'interaction est instanciée en désélectionnant tous les boutons.

Une fois définie, la dépendance peut être déclarée entre plusieurs groupes de boutons.

```
(deflink exclusion-mutuelle (:buttons)
  :var ((active? :initform #f :accessor active?))
  :behavior
    (((deselect :buttons-receiver) implies      (set! active? link #f))
     ((select   :buttons-receiver) implies      (set! active? link #t))
     ((select   :buttons-receiver) permitted-if (not (active? link)))))

(define-method action-before-effective ((lk exclusion-mutuelle) initargs)
  (for-each deselect (give lk :buttons)))

(define b1 (make button))  (define b2 (make button))
(define b3 (make button))  (define b4 (make button))
(define b5 (make button))  (define b6 (make button))
(define gp1 (make exclusion-mutuelle :buttons (list b1 b2 b3)))
(define gp1 (make exclusion-mutuelle :buttons (list b4 b5 b6)))
```

FIG. 2.3 – *Exemple d'interaction en* FLO

Pour résumer, la définition d'une interaction à l'aide d'une entité indépendante des objets sur lesquels elle s'applique permet de totalement dissocier les comportements intrinsèques des objets de la communication entre les objets. Ceci offre également une meilleure abstraction des interactions comparée aux approches exprimant les interactions dans l'interface des objets interagissants. En étant une entité de première classe, l'interaction peut « embarquer » l'intégralité de la sémantique de la communication qu'elle définit et être déclarée indépendamment des objets interagissants.

Cependant, l'absence d'héritage des comportements des interactions, que l'on retrouve chez la plupart des approches étudiées, est un facteur limitant la réutilisation des interactions puisqu'elle ne permet pas de factoriser les comportements communs à un ensemble d'interactions mais implique de les redéfinir dans chacune des interactions de cet ensemble.

2.2.2 Expression des interactions à l'aide d'un schéma de conception

Dans [SML99] les interactions sont décrites dans le même langage que les objets à l'aide d'un schéma de conception (*Design Pattern* [GHJ+95]) nommé *Dynamic Composite Adapter Pattern*. Ce schéma de conception permet de générer des *objets d'adaptation* qui correspondent, du point de vue du modèle à composants, aux connecteurs et qui vont encapsuler les objets de l'application, les composants, de manière à pouvoir les étendre dynamiquement. Ainsi la mise en œuvre de ce schéma de conception permet de définir des interactions qui pourront être instanciées dynamiquement sur les objets.

Tout comme pour les approches précédentes, les interactions sont définies en tant qu'objets et disposent par conséquent de toute la sémantique propre des objets. En encapsulant les objets de la communication, ils peuvent intercepter les appels de méthodes sur ceux-ci et ainsi modifier le comportement d'une méthode (tout en respectant les paradigmes objets d'encapsulation et de modularité). Ainsi, un connecteur peut être utilisé pour décrire les interactions entre des objets en dehors de ces objets, et ce de manière indépendante des objets interagissants.

Cependant, le schéma de conception *Dynamic Composite Adapter*, qui est mis en œuvre par le biais d'un objet, ne sait gérer que les **relations unidirectionnelles** n-aires (alors que les approches décrites ci-dessus sont bi-directionnelles). Ainsi une interaction bi-directionnelle n-aire doit être réalisée à l'aide de *n* connecteurs unidirectionnels.

Critère C3.1 : Interactions n-aires

Les interactions doivent permettre l'expression de la sémantique de la communication entre un ensemble quelconque d'objets. Ceci doit se traduire par le fait que le nombre de participants ne doit pas être restreint et qu'aucune distinction ne doit être réalisée entre les objets interagissants.

NOTE. — L'absence de distinction entre l'entité représentant l'interaction et les autres objets interagissants participants à cette interaction est une conséquence de ce critère.

Ce schéma de conception est basé sur le concept de polymorphisme des langages à objets. Il utilise le fait que changer dynamiquement le type d'un objet permet aussi de changer son comportement. Il permet ainsi d'abstraire la communication en l'encapsulant dans une classe et offre, par la même, le concept d'héritage aux interactions.

Cependant les appels aux connecteurs sont à la charge du programmeur (en changeant le type de l'objet interagissant en un type *objet d'adaptation* (dont la classe est définie à l'aide du schéma de conception *Dynamic Composite Adapter*) et donc non transparent. Ceci nécessite par conséquent d'inclure explicitement dans le code des objets qui utilisent des connecteurs le code permettant d'appeler ces connecteurs.

Malgré cette forte contrainte, le principal avantage de cette solution est de pouvoir être **mise en œuvre dans n'importe quel langage orienté objet** supportant le concept de polymorphisme puisque, comparé aux précédentes approches, aucune extension de syntaxe n'est requise.

Critère C1.3 : Indépendance vis-à-vis des concepts du langage applicatif

Le concept d'interaction doit être indépendant de tout modèle de programmation. Ceci doit se traduire par une description des interactions indépendante de tout langage de programmation applicatif et de tout concept propre à un tel langage.

NOTE. — Ceci implique que la description des interactions ne fait appel qu'aux concepts disponibles dans tous les langages cibles (langages compilés et fortement typés dans notre cas, style C++ ou Java).

2.2.3 Expression des interactions sous forme de classes de connecteurs

L'approche nommée Aspectual Components [LLM99] définit une syntaxe étendue de Java pour décrire les connexions (les composants étant les classes Java). Afin d'offrir aux connexions, qui décrivent la sémantique des interactions, un niveau d'abstraction similaire à celui des objets dans un langage orienté objet, les auteurs de Aspectual Components définissent pour les connexions une entité qui est l'équivalent de la notion de classe du paradigme objet. C'est cette entité qu'ils nomment composant. Elle représente en réalité une « classe » de composants complexes. D'ailleurs, dans [LLM99], les auteurs des Aspectual Components évoquent une possibilité de transcrire leurs composants vers de l'AspectJ [LK99].

Un composant définit un **ensemble de participants formels** à la communication, un ensemble de services requis et un ensemble de services fournis par la communication. Une interaction correspond à une instance d'un composant. Lors de l'instanciation d'un composant sont associés les participants formels aux classes participantes à la connexion ainsi que les services du composant aux méthodes des classes participantes.

Critère C1.4 : Description formelle

La définition des interactions doit être réalisée de manière formelle. Ceci doit se traduire par une description formelle des objets interagissants dans les comportements des interactions.

NOTE. — Ceci implique une phase permettant d'associer les objets interagissants à leur description formelle dans les interactions. Il s'agit d'une phase d'instanciation. Ceci est d'autant plus vrai que les interactions sont des entités de première classe (critère C1.1).

EXEMPLE. — La figure 2.4 (issue de [LLM99]) montre un exemple de composant et son instanciation permettant de réaliser une trace des accès en lecture. On peut remarquer sur cet exemple que l'interaction est entièrement décrite dans l'entité désignée par le mot-clef `component`. Ceci permet de rendre indépendant les objets interagissants des interactions définies. L'instanciation du composant est réalisée par le biais d'une entité désignée par le mot-clef `connector`. C'est cette entité qui effectue la correspondance entre le nom réel des opérateurs et leur dénomination au sein du composant.

```
package Shapes;
class Point {
  private int x = 0;
  private int y = 0;
  void set (int x, int y)
    { setX (x); setY (y); }
  void setX (int x) { this.x = x; }
  void setY (int y) { this.y = y; }
  int getX () { return this.x; }
  int getY () { return this.y; }
}
```

```
package ShowAccess;
component ShowReadAccess {
  participant DataToAccess {
    expect Object readOp ();
    replace Object readOp () {
      System.out.println("Read access on " +
                         this.toString ());
      // the expected readOp ()
      return expected ();
    }
  }
}
```

```
package Deployment;
import ShowAccess.*;
import Shapes.*;

connector ShowReadAccess {
  Point is ShowReadAccess.DataToAccess  with { readOp = get* };
}
```

FIG. 2.4 – *Exemple d'interaction avec les* Aspectual Components

Cette approche, basée sur la programmation par Aspects [KLM+97], permet de rendre la **description des connexions** (les interactions) **indépendante d'un langage de programmation donné**. De plus, les auteurs privilégient l'instanciation d'un connecteur sur le binaire des composants (le byte-code). Ceci permet d'utiliser des connexions avec des composants dont le code source n'est pas disponible et d'**ajouter les connexions à l'exécution** (grâce au mécanisme de chargement dynamique de classes de Java). Cela suppose cependant de disposer d'un outil permettant de modifier le byte-code (tel que Javassist [Chi98]).

Critère C4.1 : Dynamicité de la définition des interactions
Les interactions doivent pouvoir être définies et déclarées dynamiquement lors de l'exécution de l'application sans nécessiter l'arrêt de cette dernière. Ceci doit se traduire par la présence d'un mécanisme permettant de modifier, d'ajouter, de supprimer dynamiquement des interactions dans un système.

Critère C1.3' : Indépendance vis-à-vis des langages de programmation
La description des interactions doit être indépendante d'un langage de programmation donné. Ceci doit se traduire par la définition d'un langage dédié à la description des interactions (un langage d'aspects).
NOTE. — Ce critère peut être considéré comme une généralisation du critère C1.3.

Pour résumer, cette approche permet de manipuler les interactions et les objets de manière identique. Elle respecte les principes du modèle sous-jacent (encapsulation, héritage, etc.) et permet une définition incrémentale des interactions sous forme de schémas d'interactions (appelés *composants*). De plus, la description des interactions est indépendante d'un langage à objets donné. Cependant les connecteurs ne peuvent être définis de manière incrémentale par raffinement.

2.2.4 Expression des interactions par une entité centralisant la coordination

Les approches Jada [CR97], JATLite [Jeo98] et Agent Management System [JC99] définissent un système multi-agents [Fer95] dont l'un des agents centralise la coordination de toutes les communications entre agents.

Les approches Jada et JATLite sont des bibliothèques Java [CH96] permettant le développement d'agents en Java. Grâce à Jada, les applications Java distribuées peuvent accéder à un espace de partage d'objets (un *space object*) à des fins de partage de données et de coordination. Jada définit un processus comme étant une séquence d'actions exécutée par un agent et par conséquent chaque *thread* est, pour Jada, un agent. Un espace de partage permet aux différents agents de coordonner leurs communications en s'échangeant des objets. C'est donc lui qui décrit la sémantique des interactions. Ainsi, Jada utilise la sémantique de Java pour la communication entre les agents.

JATLite et *Agent Management System* (AMS) proposent une gestion de la communication différente de celle de Jada. En effet, ils définissent un outil de communication des messages basé sur le langage *Knowledge Query and Manipulation Language* (KQML) [F$^+$92, LF93, LF98] ainsi qu'une entité permettant de coordonner les interactions entre agents. Cette entité est nommée agent de routage en JATLite et gestionnaire d'agents en AMS. Les messages qui doivent être transmis à des agents sont stockés dans une file de messages qui sera traitée par l'agent de routage ou le gestionnaire d'agents.

Ainsi, l'entité de coordination reçoit des requêtes (les messages) provenant d'agents de l'application, détermine le ou les agents capables de résoudre la requête, les invoque en leur envoyant un message en KQML décrivant la requête. Une fois la requête traitée son résultat est expédié à l'agent émetteur. Il est à noter, cependant, qu'avec AMS les agents ne sont pas obligés d'utiliser le gestionnaire d'agents pour communiquer entre eux. Dans ce cas les agents ne bénéficient pas des possibilités de coordination de messages offertes par le gestionnaire d'agents.

Critère C3.2 : Aucune discrimination entre les objets au sujet du support des interactions
Le support des interactions doit être le même pour tous les objets. Ceci doit se traduire par la possibilité de définir et de déclarer des interactions sur n'importe quel objet du système sans imposer de contraintes sur ce dernier.
NOTE. — Ce critère peut être considéré comme une extension du critère C2.1 (transparence du support des interactions).

Ces trois approches décrivent la sémantique des communications coordonnées entre les agents par une unique entité : l'*object space* pour Jada, l'agent de routage pour JATLite et le gestionnaire d'agents pour AMS (figure 2.5). Cette entité centralise et définit la sémantique de toutes les communications coordonnées inter agents (les interactions). Il est donc nécessaire de la redéfinir pour modifier la sémantique des interactions existantes entre les agents. JATLite joue ici sa carte maîtresse puisqu'il permet de redéfinir la sémantique des communications entre agents à l'aide du langage KQML sans avoir à réécrire l'agent de routage.

Cependant, dans tous les cas, le couplage entre les agents et les interactions est assez fort puisque le comportement de l'agent gérant les communications doit être adapté aux agents qui s'exécutent. De plus aucune de ces approches ne permet l'ajout ou la modification dynamique des interactions (cela nécessiterait une substitution de l'entité de coordination par une autre intégrant les nouvelles interactions lors de l'exécution).

La transparence de l'exécution des interactions n'est pas nécessairement vérifiée. En effet, les interactions se trouvent dans l'entité centralisant la coordination. Or celle-ci ne gère pas les communications non coordonnées. Dans JATLite, par exemple, un agent doit s'enregistrer auprès de l'agent de routage pour pouvoir bénéficier de la coordination des messages qui lui sont passés. Ainsi les interactions ne sont transparentes que pour les agents qui se sont enregistrés auprès de l'agent de routage. Nous avons une situation similaire en AMS ou seules les communications entre agents utilisant l'architecture AMS pourront être coordonnées et donc disposer d'une exécution transparente des interactions dont elles sont la source.

En fait, ces approches permettent de séparer le comportement de coordination des agents des autres comportements en le centralisant dans une entité spécifique afin de simplifier l'élaboration d'applications distribuées hétérogènes. Cette entité dispose de sa propre sémantique pour l'exécution des communications entre les agents dont elle responsable.

Cependant, cette entité peut très vite devenir un **goulot d'étranglement**. De plus, la sémantique qu'elle définit est la même pour toutes les communications (la modification de cette sémantique étant limitée). Ces approches offrent donc un support très restreint des interactions telles que nous les définissons. De plus, la durée de vie des interactions est égale à celle de l'application.

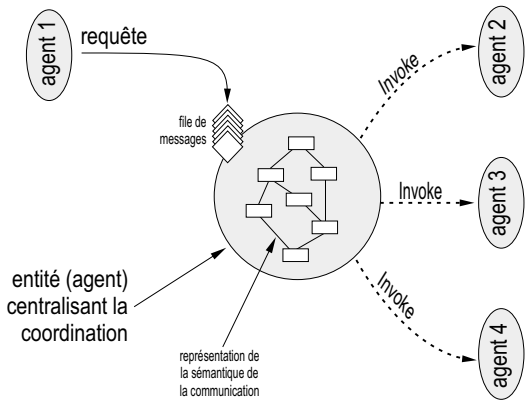


FIG. 2.5 – *Vue d'ensemble des approches centralisant la coordination*

Critère C3.3 : Décentralisation du support des interactions

La gestion des interactions (et notamment leur exécution) ne doit pas être basée sur une architecture centralisée mais, au contraire, sur une architecture « répartie ». Ceci doit se traduire par la capacité de chaque interaction à pouvoir exécuter, de manière autonome, la sémantique de la communication qu'elle décrit.

NOTE. — Ce critère implique que la cohérence du graphe des interactions n'est pas nécessairement gérée par le système (puisque cela implique une vision globale du système par une seule et même entité) comme cela est le cas pour les approches étudiées dans ce paragraphe.

2.2.5 Synthèse partielle

Le tableau 2.2 présente une synthèse du support des interactions par les approches étudiées dans ce paragraphe les plus représentatives.

Critères														
Abstraction des interactions				Intégration dans le modèle à objets				Expressions des interactions				Aspects dynamiques		
C1.1	C1.2	C1.3	C1.4	C2.1	C2.2	C2.3	C2.4	C3.1	C3.2	C3.3	C3.4	C4.1	C4.2	C4.3
Synchronizers [FA93]														
✓	✓	✗	✓	✓	✓	✓	-	✓	✗	✓	✗	✗	✗	✗
FLO [Duc97b]														
✓	✓	✗	✓	✓	✓	✓	✓	✓	✘	✓	✓	✘	✗	✗
Dynamic Composite Adapter Pattern [SML99]														
✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✗	✗	✗
Aspectual Components [LLM99]														
✓	✓	✓	✓	✓	✓	✓	✘	✓	✗	✓	✓	✘	✗	✗
JATLite [Jeo98]														
✓	✗	✗	✓	✘	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗

✓ : supporté, ✗ : non supporté, ✘ : supporté sous certaines conditions

TAB. 2.2 – *Synthèse du support des interactions par le biais d'une entité spécifique*

2.3 Support des interactions grâce à un langage spécifique

Toutes les approches étudiées dans le paragraphe précédent utilisent le même langage à objets pour décrire les interactions et les objets de l'application. Ceci soit par extension du modèle à objets, soit par utilisation combinée de plusieurs concepts du modèle à objets. Il existe des approches qui décrivent les interactions à l'aide d'un langage spécifique différent du langage à objets servant à décrire les applications. Ce paragraphe présente ces approches.

Les langages de type *Architectural Description Language* (ADL) [Med97], *Module Interconnection Language* (MIL) [PDN89], ou *Interface Definition Language* (IDL) [Lam87], issus des récentes recherches de la communauté des architectures logicielles [Gar95, GPT95, Wol96, MP98], sont largement utilisés pour décrire les interactions dans les approches à composants [EHT95, SML99], les langages de type *Agent Communication Language* (ACL) [LFP99] l'étant pour les approches à agents.

2.3.1 Expression des interactions à l'aide d'un langage de type ADL

Parmi les approches exprimant les interactions à l'aide d'un ADL, on peut citer Conic [MKS89], REX [KMS+92], Weaves [GR91, GQ94], POLYLITH [Pur94], Regis [MDK94], UNICON [SDK+95, SDZ96], OLAN [BAB+96, BR96], Darwin [MDE+95, MK96], RAPIDE [Bry94, LKA+95, Luc96], C2 [MOR+96, MRT99], les Policies [LS97], les travaux de G. AGHA sur les *Interaction Patterns* [Agh97], Wright [AG97, All97, ADG98], Traversal [OW99] et CoLaS [CD99].

Toutes ces approches décrivent l'interface des composants et des connecteurs (qui contiennent la sémantique des interactions) à l'aide d'un langage de description (OCL pour OLAN, Darwin pour Regis, ICA pour RAPIDE par exemple). La définition des interactions dans les connecteurs permet de clairement les séparer des fonctionnalités intrinsèques de l'application (contenues dans les composants) et de respecter les paradigmes sous-jacents.

Mis à part Conic, REX, OLAN et UNICON, ces approches permettent de définir de **nouvelles sémantiques de communication** sous la forme de connecteurs. Ces nouveaux types de connecteurs sont soit définis intégralement dans le langage déclaratif (RAPIDE), soit leur interface est décrite dans le langage déclaratif et leur mise en œuvre déportée dans un langage à objets traditionnel (Regis et Darwin par exemple). Cette dernière solution permet de disposer de la puissance d'expression d'un langage à objets lors de la définition des interactions.

Les approches OLAN et UNICON, quant à elles, ne proposent qu'un ensemble prédéfini et non extensible de connecteurs, chaque connecteur décrivant une sémantique de communication (synchrone ou asynchrone, basée sur RPC [BN84], etc.). Dans UNICON, cette sémantique est décrite sous la forme de listes de propriétés et d'attributs non modifiables. Ainsi ces approches ne permettent de définir que des interactions dont la sémantique de communication est proposée par l'un des types de connecteurs. De ce fait si, par exemple, aucun connecteur ne gère la communication avec le bus logiciel de CORBA, il ne sera pas possible de décrire une interaction utilisant ce protocole. Cette contrainte limite ainsi, de manière plus ou moins importante, la puissance d'expression des interactions.

Critère C3.4 : Extension des sémantiques de communication
L'ensemble des sémantiques de la communication (synchrone, asynchrone, etc.) utilisées pour décrire les interactions doit être extensible et modifiable.

NOTE. — Une conséquence de ce critère est la possibilité de changer les protocoles de communication sous-jacent aux sémantiques utilisées dans la description des interactions.

EXEMPLE. — La figure 2.6 page suivante (issue de [BR96]) décrit en OCL un composant primitif modélisant un compte bancaire ainsi qu'un composant complexe modélisant un virement entre deux comptes bancaires à l'aide d'un connecteur.

La description du composant primitif est décomposée en trois éléments : l'un indiquant où trouver la mise en œuvre effective du composant et ses caractéristiques (du code C++ [Str91] dans cet exemple), un autre fournissant une interface OCL pour le composant et un troisième permettant d'adapter le composant « physique » (en C++) avec l'interface OCL. La description du composant complexe se compose uniquement d'une interface OCL décrivant l'intégralité des comportements du composant et spécifiant les connexions (c'est-à-dire les interactions) à utiliser.

Il est à noter que la description des connexions et des composants complexes à l'aide d'un langage déclaratif implique une première compilation vers un langage de programmation traditionnel (en général le langage des composants primitifs). Ainsi, en raison de cette phase de compilation, seules quelques approches sont capables de modéliser des architectures ayant un comportement dynamique [ADG98] et donc permettant de définir de nouvelles interactions au cours de l'exécution.

Les plus remarquables d'entre elles sont RAPIDE, Darwin (et indirectement Regis), les *Interaction Patterns* et Conic. Dans le cas de RAPIDE, le langage ICA propose des clauses *where* permettant de

```
class 'C++' Account {
  name: 'account.cpp';
  path: '/project/exemple/';
  compiler: 'g++';
  ...
};

management AccountAdm : AccountImpl
  { Node.os == 'AIX4' };

component Account {
  implementation AccountImpl;
  management AccountAdm;
};

implementation BankOffice :
  BankOfficeItf uses AccountItf,
                     TrnsferAgtItf {
  theAccount = instance AccountItf;
  theManager = instance TrnsferAgtItf;
  ...
  theManager.Transfer (amount) =>
    theAccount.Deposit (amount),
    theAccount.Withdrawal (amount)
    using moneyTransfer;
  ...
};
```

```
primitive implementation AccountImpl :
  AccountItf uses Account {
  AccountNumber => Account.acct_Id;
  Deposit       => Account.Add
                     (in float amount);
  Withdrawal    => Account.Substract
                     (in float amount);
};

interface AccountItf {
  attribute long AccountNumber;
  provide   Deposit (in float amount);
  require   Withdrawal (in float amount);
};

connector moneyTransfer uses Proc_Call {
  inputs  IN[1,1];
  outputs OUT[2,2];
  routing {
    on IN[1] (x) select {
      if x > 0: OUT[1] (x)
          else: OUT[2] (x)
    }
  }
};
```

FIG. 2.6 – *Exemple d'interaction avec* OLAN

redéfinir des connexions lors de l'exécution à l'aide des opérateurs *link* et *unlink*. Il offre de plus un point de vue orienté objet : de nouveaux composants architecturaux peuvent être définis puis instanciés pour autant que l'on puisse les créer, lors de l'exécution de l'application, grâce à un langage orienté objet [LKA+95].

Le langage de Darwin permet seulement de définir l'aspect *structurel* de l'architecture [MK96] grâce à l'utilisation de scripts interprétés permettant l'instanciation, la suppression et la modification dynamique de liens entre composants et connecteurs. Il se base sur l'algèbre du π-calcul [Mil92] pour donner une sémantique à la reconfiguration. « *Their use of the Pi-Calculus to give semantics to reconfiguration is elegant and suggestive of the power of a more flexible "dynamic" process algebras.* » [ADG98]. L'approche des *Interaction Patterns* définie un protocole à métaobjets (MOP) présent à l'exécution. Ce MOP permet de dynamiquement modifier le mécanisme spécifiant la sémantique des interactions, c'est-à-dire le mécanisme de gestion des boîtes aux messages des acteurs (en effet, cette approche se focalise sur les langages d'acteurs).

Critère C4.2 : Configuration dynamique

Les interactions doivent pouvoir être configurées dynamiquement. Ceci doit se traduire par la présence à l'exécution d'un mécanisme permettant de modifier, d'ajouter ou de supprimer les définitions formelles des interactions dans un système.

Dans Conic, une toute autre solution a été utilisée pour le support de la configuration dynamique du système. Elle s'apparente au **concept de dépôt d'interfaces** de CORBA [IR01]. En effet, la configuration « en cours » des communications (c'est-à-dire les interactions) est stockée dans une base de données disponible en ligne [KM85]. La consultation de cette base de données pour obtenir des informations sur la configuration actuelle est accessible par toutes les applications du système. Celles-ci peuvent ainsi reconfigurer le système en mettant à jour la base de données.

Critère C4.3 : Dépôt des définitions formelles des interactions

Les définitions formelles des interactions doivent pouvoir être consultées, en vue de modification, d'ajout ou de suppression, lors de l'exécution d'un système. Ceci doit se traduire par la mise en place d'un mécanisme de dépôt des définitions formelles des interactions.

NOTE. — Dans un environnement distribué, ce dépôt doit être accessible en ligne depuis n'importe quel site du système.

EXEMPLE. — La figure 2.7 page suivante présente un exemple d'interaction à l'aide de Conic. Il s'agit d'un moniteur de température et de pression. La colonne de gauche décrit, dans une version étendue de Pascal[2] [KMS+84] le code d'un module (l'équivalent de la classe dans le modèle à objets). La colonne de droite décrit, dans le langage de configuration de Conic [DKM+84], les créations des objets et les liens (interactions) existant entre ces objets.

Il est à noter que la description des interactions est réalisée sous la forme d'un module. Ainsi, ce module peut être instancié plusieurs fois et connecté à d'autres modules par des modules décrivant des interactions. En fait, les concepteurs de Conic considèrent les modules définissant les différents processus de l'application comme des nœuds logiques dans un graphe et les modules définissant les interactions comme les arêtes de ce graphe.

2. Les concepts de module et de mode de communication (synchrone, asynchrone, etc.) sont notamment ajoutés. De plus, un module peut être mis en œuvre à l'exécution sous la forme d'une tâche (processus).

```
task module scale (scalefactor: integer);
  entryport
    control: boolean;
    input: real reply signaltype;
  exitport
    output: real reply signaltype;
  var
    value: real;
    active: boolean;
begin
  active := false;
  loop
    select
      receive active from control
    or
      when active
      receive value from input
          reply signal =>
        send value/scalefactor
          to output wait signal;
    end
  end
end.
```

```
group module monitor (Tfactor,Pfactor: integer);
  exitport
    press,temp: real reply signaltype;
  entryport
    control:boolean;
  use
    scale; sensor;
  create
    temperature: sensor;
    pressure: sensor;
    Tscale: scale (Tfactor);
    Pscale: scale (Pfactor);
  link
    temperature.output to Tscale.input;
    pressure.output to Pscale.input;
    Tscale.output to temp;
    Pscale.otput to press;
    control to TScale.control, Pscale.control;
end.
```

FIG. 2.7 – *Exemple d'interaction en Conic*

L'hétérogénéité des plates-formes est supportée par toutes ces approches. Par contre, **l'hétérogénéité des langages n'est pas toujours supportée**. Par exemple, les composants de Regis sont nécessairement des objets C++. De même, la description des applications de Conic est limitée à une version étendue du langage Pascal.

Finalement, toutes les approches offrent une transparence de l'exécution des interactions. En effet, les interactions sont définies dans les connecteurs, or la sémantique de la communication des composants entre eux par le biais des connecteurs est directement spécifiée par le modèle à composants.

Pour résumer, ces approches dissocient clairement les objets interagissants des interactions qui sont définies dans des entités nommées connecteurs. Ces connecteurs sont décrits dans un langage différent de celui du langage applicatif. Ces approches offrent donc une abstraction des interactions suffisamment importante pour permettre la description des interactions indépendamment d'un langage à objets donné. Une conséquence de ce point fort est la nécessité, pour la plupart des approches, « d'embarquer » les interactions de manière statique et de n'offrir aucun support à leur évolution durant l'exécution.

2.3.2 Expression des interactions à l'aide d'un langage de type ACL ou de logique du premier ordre

Certaines approches, généralement basées sur les agents, définissent un langage spécifique pour décrire les sémantiques de communications entre agents. Il s'agit d'un généralement langage de type *Agent Communication Language* (ACL) [LFP99]. C'est le cas de l'approche COOL [BF95] et de Reactive C [Bou91] ainsi que de l'approche définie dans [Sin00]. D'autres approches utilisent des langages basés sur la logique du premier ordre ou sur les processus algébriques. C'est le cas de *Multi-Agent Framework* (MAF) [BCJ+95, BPC+95] et des travaux de P. CIANCARINI autour de PoliS [Cia91, CFM98, CFM00].

Les approches COOL, Reactive C, Junior, et celle définie dans [Sin00] modélisent les activités de coordination (les interactions) par le biais de graphes. Pour COOL, il s'agit d'une machine à états finis (*Finite State Machine*, FSM) et d'un ensemble de règles de conversation (*conversation rules*) décrivant le raisonnement pour déterminer la prochaine arête à suivre dans le graphe représentant la FSM. Avec Reactive C et Junior il s'agit d'automates, tandis que dans l'approche de [Sin00] il s'agit de l'extension de Parunak des graphes de Dooley [Par96]. MAF, une extension de *An Open Agent Architecture* (OAA) [CCW+94], se base sur la sémantique des prédicats logiques du premier ordre (*first order predicate logic*) pour décrire les interactions. Ainsi, les activités de coordination sont définies en ICL [BPC+95], un langage dont la syntaxe est similaire à Prolog [SS94]. Les travaux de P. CIANCARINI utilisent PoliS [CFM98] pour décrire les interactions. Ce langage est basé sur les concepts des processus algébriques et sur les notions d'espaces partagés nommés *tuple space*.

Pour toutes ces approches, **la définition des interactions est statique**. De plus, les graphes ou les prédicats utilisés pour décrire la coordination sont associés aux agents pour lesquels ils définissent la sémantique de communication. De ce fait, **les interactions sont fortement liées aux agents** :« *Agents communicating with each other require a well-known set of conventions. This set of conventions comprises a protocol that must be implemented at both ends of a connection* » [BPC+95]. Ceci implique qu'il n'est pas possible de définir dynamiquement de nouvelles interactions.

La définition statique des interactions a deux autres conséquences. La première conséquence, un avantage en réalité, est la **transparence de l'exécution des interactions**. En effet, chacune de ces approches intègre un support du langage de description des interactions directement dans la sémantique de communication entre agents du système multi-agents. La seconde conséquence concerne la durée de vie des interactions. Leur création et leur destruction est, en effet, liée à la création et à la destruction de l'agent qui les définit.

Il est à noter que l'approche MAF permet l'ajout, la suppression ou le remplacement d'un agent à tout moment sans impliquer de changement dans les autres agents. Ainsi, elle rend possible la modification de la sémantique de communication associée à un agent (par son remplacement).

EXEMPLE.— La figure 2.8 montre un exemple (issu de [BCJ+95]) d'expression des interactions par le biais d'un langage spécifique. L'environnement utilisé est MAF. Les sémantiques de communications (c'est-à-dire les interactions) sont décrites à l'aide des routines `do_event`. Ces interactions sont définies dans le prédicat `solvable`. Comme on peut le voir, ces définitions sont statiques et décrites avec l'agent. Ainsi, les interactions sont très fortement liées à l'agent et ne peuvent être dynamiquement modifiées.

```
:-[agent].
agentName('calendar').
solvable( [
           get_free_time_slot (_Date, _Person, _FreeTime),
           where (_Person, _Place, _WithPerson)
          ] ).

do_event(_, get_free_time_slot (Date, Person, FreeTime)) :-
         get_appointment_file (Person, CalendarFile),
         get_free_time_C (CalendarFile, Date, FreeTime), !.

do_event(_, where(Person, Place, WithPerson)) :-
         get_appointment_file (Person, CalendarFile),
         get_current_date_and_time (Date, Time),
         get_appointment (CalendarFile, Date, Time, Appointment),
         parse_appointment (Appointment, WithPerson, Place).
...
```

FIG. 2.8 – *Exemple d'interaction avec l'approche* MAF

L'approche Brainstorm/J

Brainstorm/J [ZA00] est un *framework* Java pour agents intelligents. Nous le présentons en marge des autres approches car il est basé à la fois sur un ensemble de composants et sur un langage de description des interactions. L'architecture de Brainstorm/J repose sur le fait qu'un système multi-agents peut être considéré comme un système orienté objet associé à un méta-système [AP97, AP98]. Elle est mise en œuvre en JavaLog, un langage multi-paradigme basé sur Java et Prolog [AZI99].

Ainsi, un agent est considéré comme un objet disposant d'une couche comportementale intelligente, son métaobjet. Cette couche contient notamment la sémantique de la communication entre agents. De ce fait, un agent est composé d'un objet et d'un ensemble de métaobjets. Brainstorm/J propose plusieurs métaobjets de gestion de la communication. Chacun de ces métaobjets permet de décrire la sémantique de la communication à l'aide d'un ACL tel que KQML [FFM+94, LF97] ou FIPA [FIP97, FIP98]. De plus, les agents disposent d'une gestion des connaissances. Ces connaissances permettent de définir un **ensemble de réactions pour un événement donné** et sont décrites à l'aide d'un langage proche de Prolog.

Dans cette approche, la communication entre agents peut être décrite de plusieurs façons différentes : à l'aide d'un langage ACL, d'un métaobjet ou de « connaissances événementielles ». Par conséquent, ceci peut rendre beaucoup plus complexe la mise en œuvre d'une application. Cependant, en contrepartie, cela offre un **mécanisme flexible et extensible de description des interactions**. En effet, il est possible d'ajouter de nouveaux métaobjets de communication.

Il est à noter que la description des communications étant réalisée par les métaobjets la gestion et l'exécution des interactions est transparente du point de vue du programmeur. Cependant les interactions ont la même durée de vie que les agents. Ceci est d'ailleurs une constante pour toutes les approches à agents étudiées.

2.3.3 Expression des interactions sous forme de méta-programmes

Les approches Object Communities [CM93], CodA [McA95] et Connectors in Open Language [ALP99] utilisent explicitement les techniques de méta-programmation et de protocole à métaobjets (MOP) [BKdR91] pour décrire les interactions. Celles-ci sont définies dans les méta-programmes. La sémantique des interactions est « fusionnée » avec les comportements intrinsèques des objets de l'appli-

cation lors de la transformation de l'application par exécution du méta-programme. En fait, **les interactions redéfinissent les méta-comportements de la communication** (et donc de l'envoi de messages).

Critère C2.3 : Utilisation des concepts de communication du modèle sous-jacent
Les interactions ne doivent pas définir de nouveaux concepts de communication mais, au contraire, utiliser ceux du modèle sous-jacent. Ceci doit se traduire par la redéfinition (ou plus exactement l'extension) du concept d'envoi de messages du modèle à objets.

Cette solution permet de clairement dissocier les objets (écrits au niveau de base) des interactions (écrits au niveau méta) et de pouvoir simplement et facilement ajouter ou enlever des interactions (par application ou non des méta-programmes). Un méta-programme étant avant tout un programme, bien souvent écrit dans le même langage que les autres objets du système, les interactions sont donc des objets. Ainsi il est possible de définir une interaction comme étant la composition (par héritage) d'autres interactions.

Cependant, les méta-programmes ont accès à la partie privée des objets et donc **les interactions peuvent ne pas respecter le concept d'encapsulation des objets**, rendant les interactions dépendantes de la mise en œuvre des objets interagissants.

L'appel des interactions (les méta-programmes) est entièrement géré par le système (lors de la compilation dans le cas d'un MOP statique ou au cours de l'exécution pour un MOP dynamique) et donc l'exécution d'une interaction est transparente du point de vue du programmeur.

La présence d'un MOP entraîne inévitablement un sur-coût, non négligeable, résultant de la réification de certains mécanismes du langage [MMW+92]. Pour les Object Communities, cette réification, et donc le sur-coût qui y est associé, a lieu durant l'exécution et porte sur l'envoi de messages. Ainsi le contrôle de l'exécution est donné aux méta-programmes (et donc aux interactions), à chaque appel de méthode sur un objet, en réifiant cet appel.

Lorsque le MOP est statique (cas de [ALP99]), le sur-coût a lieu durant la compilation, et les performances de l'application ne sont pas pénalisées par des mécanismes de réification à l'exécution. En contre-partie les interactions (c'est-à-dire des méta-programmes) ont la même durée de vie que l'application et le programmeur de ces interactions doit écrire lui même le code permettant de les activer et de les désactiver en fonction de l'état de l'application.

Ainsi la solution consistant à décrire les interactions sous forme de méta-programmes à l'exécution permet, comparée à celle basée sur des méta-programmes de compilation, une dynamicité de la déclaration (instanciation) des interactions en échange d'une **dégradation des performances globales de l'application** en raison de réifications lors de l'exécution.

Critère C2.4 : Conservation des performances initiales
Le support des interactions ne doit pas dégrader sensiblement les performances globales du système. Ceci doit se traduire notamment par l'utilisation de mécanismes « *poids mouche* »*.*

2.3.4 Synthèse partielle

Le tableau 2.3 page suivante présente une synthèse du support des interactions par les approches étudiées dans ce paragraphe les plus représentatives.

2.4 Support des interactions par le biais d'autres mécanismes

Bien que le support des interactions soit généralement réalisé par le langage applicatif, par une entité spécifique ou par l'utilisation d'un langage approprié, d'autres mécanismes existent. Ils sont présentés dans ce paragraphe.

2.4.1 Expression des interactions à l'aide de composants

Dans certaines approches, les agents sont définis par le biais de composants. C'est par exemple le cas des approches Java Agent Framework [Hor98] et MALEVA [Lhu98]. Elles définissent chaque agent sous la forme d'un ensemble de composants, un composant décrivant un comportement de base de l'agent.

Dans MALEVA [Lhu98], un agent est une entité définie par un comportement, par un ensemble de bornes[3] d'entrées, par un ensemble de bornes de sorties, et par un gestionnaire de messages (ce

3. MALEVA dissocie les flots de données des flots de contrôle. Ainsi les bornes sont de deux types : celles pour les données et celles, pour le contrôle.

Critères														
Abstraction des interactions				Intégration dans le modèle à objets				Expressions des interactions				Aspects dynamiques		
C1.1	C1.2	C1.3	C1.4	C2.1	C2.2	C2.3	C2.4	C3.1	C3.2	C3.3	C3.4	C4.1	C4.2	C4.3
Darwin [MK96]														
✓	✓	✓	✓	✓	✓	✓	-	✓	✗	✓	✓	✗	✓	✗
OLAN [BR96]														
✓	✓	✓	✓	✓	✓	✓	-	✓	✗	✓	✗	✗	✗	✗
RAPIDE [Bry94]														
✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✗	✗
Conic [MKS89]														
✓	✓	✗	✓	✓	✓	✓	-	✓	✗	✓	✗	✓	✓	✓
COOL [BF95]														
✓	✗	✗	✗	✓	✓	✓	-	✓	✗	✓	✗	✗	✗	✗
MAF [BPC+95]														
✗	✗	✗	✓	✓	✓	✓	-	✓	✗	✓	✗	✗	✗	✗
Brainstorm/J [ZA00]														
✗	✗	✗	✓	✓	✓	✓	-	✓	✗	✓	✓	✗	✗	✗
CODA [McA95]														
✓	✓	✓	✓	✓	✗	✓	✗	✓	✗	✓	✗	✗	✗	✗

✓ : supporté, ✗ : non supporté, ✗ : supporté sous certaines conditions

TAB. 2.3 – *Synthèse du support des interactions par le biais d'une entité spécifique*

gestionnaire simule la file des messages des agents et décrit la sémantique des interactions). De ce fait, lorsqu'un agent doit exécuter un message sur un autre agent il dépose le message à exécuter dans la boîte aux messages de l'agent receveur en utilisant le concept des connecteurs du modèle à composants. Cette sémantique est décrite par les gestionnaires de messages des agents émetteur et récepteur.

On retrouve de manière très similaire cette architecture dans *Java Agent Framework* (JAF) [Hor98], un framework étendant le modèle des Java Beans [SUN97] (le modèle d'architecture de composants de Sun Microsystem). JAF utilise de manière intensive la notion de flots d'événements du modèle Java Beans, ce qui lui permet d'instancier dynamiquement des interactions entre des composants producteurs et des composants consommateurs. Il propose en fait des mécanismes pour spécifier et résoudre les dépendances entre agents. Ainsi le support des interactions par les agents de JAF est le même que celui proposé par l'architecture Java Beans (décrit dans le paragraphe 2.1.3).

Avec ces approches, la spécification de la sémantique de la communication est définie, pour partie, du coté de l'émetteur et, pour autre partie, du côté du receveur. Il y a donc **morcellement de cette sémantique**, et, par conséquent, des interactions. Par exemple, dans MALEVA « *le gestionnaire de messages est responsable de la vision externe du composant* » [Lhu98, p. 58].

Ce gestionnaire de messages a donc en charge la gestion des bornes d'entrées et de sorties. Ainsi, il décrit la sémantique de la file d'attente associée à l'ensemble des composants décrivant l'agent. La sémantique de la communication est donc la combinaison des sémantiques définies dans les gestionnaires de messages des objets émetteur et receveur. Par conséquent, les interactions font partie intégrante des agents et ont donc une durée de vie identique à celles des agents.

Critère C3.3' : Décentralisation du support des interactions
L'expression des sémantiques d'une interaction ne doivent pas être éparpillées dans différentes entités ou en divers endroits dans du code mais, au contraire, regroupées.

On peut noter que ces approches ne permettent pas de définir dynamiquement de nouvelles interactions. De plus, elles proposent un ensemble non modifiables de sémantiques de communication. Par exemple, MALEVA propose les sémantiques de communication synchrone, asynchrone, et asynchrone bufferisée tandis que JAF utilise les sémantiques fournies par le modèle des Java Beans. La définition de connecteurs explicites décrivant l'intégralité de la sémantique de communication n'est donc pas supportée par ces approches.

EXEMPLE.— Cet exemple est issu de [Lhu98, chap. IV]. Il décrit un agent *Prédateur* composé de plusieurs composants simples (dont certains sont des agents). Le comportement de ce prédateur est le suivant : il suit une proie, fuit les autres prédateurs et, lorsqu'il n'y a ni proie ni prédateur se déplace aléatoirement.

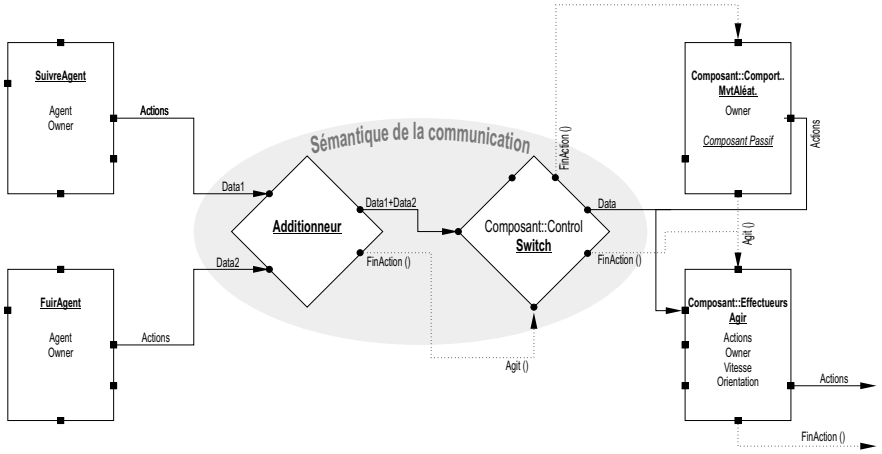


FIG. 2.9 – *Exemple d'interaction avec* MALEVA

Cet agent est composé de composants *Comportementaux* (boîtes rectangulaires) et de composants de *Contrôle* (boîtes losanges). Ces derniers composants participent à la description de la sémantique de communication entre les composants comportementaux. Ainsi, dans cet exemple, la sémantique des interactions est décrite de manière statique par les deux composants de Contrôle (figure 2.9).

2.4.2 Expression des interactions à l'aide d'un bus logiciel

Ces dernières années, le concept de bus logiciel (*Object Request Broker*, ORB) a connu un essor important, notamment avec CORBA (dont le support des interactions est décrit dans le paragraphe 3.1). Parmi les ORB existants non basés sur CORBA, certains offrent un support explicite aux interactions. C'est notamment le cas de Tj [McA96], de Jonathan [DTH+98], de FlexiNet [HHD98], de l'architecture réflexive de [BCR+98], de JavaPod [BR99] ou des *Enterprises Java Beans* (EJB) [MH99].

Toutes ces approches sont plus ou moins dérivées du modèle ODP [ODP95] qui est un modèle pour décrire un bus logiciel modulaire et extensible. Selon ce modèle, toute application distribuée peut se décomposer entre les objets et les objets de liaison (figure 2.10). Les objets de liaison sont des objets comme les autres. Il sont cependant spécialisés dans les communications entre les autres objets. Ainsi, ces bus logiciels séparent l'expression de la sémantique de la communication de celle des fonctionnalités intrinsèques des objets.

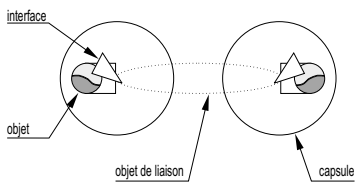


FIG. 2.10 – *Modèle ODP*

Les bus logiciels Jonathan et JavaPod décrivent les objets de liaison (c'est-à-dire les interactions) par des entités nommées *bindings* (Jonathan) ou *connecteurs* (JavaPod). Ces entités expriment à la fois le processus d'association et d'interconnexion d'un ensemble d'objets en accord avec un protocole de communication spécifique.

Les autres approches que sont Tj, FlexiNet ou l'architecture définie dans [BCR+98], utilise la réflexivité (à divers degrés) pour rendre configurable le bus logiciel et décrivent les objets de liaison par des métaobjets. Ces descriptions (par le biais de connecteurs ou de la réflexivité) des objets de liaison impliquent une **définition statique** des interactions (c'est-à-dire lors de la phase de conception). Cependant, **leur instanciation peut être dynamique**.

EXEMPLE. — Le bus logiciel Jonathan définit le concept de fabriques de liaisons (*binding factory*). Ces fabriques servent à créer dynamiquement les liaisons (*bindings*). Chaque fabrique de liaisons permet la création d'un type de liaison, c'est-à-dire d'une sémantique de communication et donc d'un « modèle » d'interaction. De ce fait, pour chaque comportement interagissant (et donc pour chaque interaction), il est nécessaire de définir une fabrique de liaisons.

Or, ces fabriques de liaisons sont des classes Java. Par conséquent, il est possible de définir de nouvelles fabriques de liaisons qui pourront être chargées dynamiquement (propriété du langage Java). Cependant, le code de ce chargement de classes est à la charge du programmeur.

Il est à noter que toutes ces approches permettent la définition de nouveaux protocoles de communication. Ceci permet la réalisation de passerelles entre différents bus logiciels (et pas seulement ceux présentés ici). Ainsi, l'interopérabilité entre l'un des bus logiciels décrit dans ce paragraphe et un autre bus logiciel peut être réalisée par la définition d'un objet de liaison spécifique.

EXEMPLE. — Les auteurs de JavaPod ont définis des connecteurs (ce sont des objets de liaison) permettant de connecter un objet client JavaPod avec un objet serveur CORBA (ou Java RMI) et inversement. Dans le premier cas, les connecteurs contiennent l'identité CORBA de l'objet serveur ainsi qu'une description du protocole nécessaire pour communiquer avec l'objet CORBA (le protocole IIOP en l'occurrence).

Le rôle des objets de liaison consiste uniquement à permettre une communication entre des objets distants (c'est-à-dire contenus dans des capsules différentes). Ainsi, les objets contenus dans une même capsule ne communiquent pas via le bus logiciel et, par conséquent, la sémantique des communications n'est pas prise en compte dans ce cas.

Critère C3.2' : Indépendance de la localité physique des objets interagissants
Le support des interactions doit être identique que les objets interagissants soient locaux ou distants. De même, aucune distinction ne doit être faite entre des objets non prévus pour être distribués et des objets supportant les appels distants (objets distribués).

Pour résumer, ces approches décrivent les interactions par le biais d'objets de liaisons (des connecteurs ou des métaobjets, suivants les bus logiciels). Elles supportent l'instanciation dynamique des interactions mais ne permettent pas leur définition dynamique. Le fait que ces approches acceptent la définition de nouveaux protocoles de communication permet une interopérabilité entre ces bus logiciels et d'autres bus logiciels.

2.4.3 Expression des interactions à l'aide de la notion de groupe d'objets

Beaucoup d'architectures distribuées gérant la résistance aux pannes sont structurées sous la forme d'un ensemble de groupes de processus coopérant entre eux par la diffusion de messages (*multicasting messages*). La construction de telles architectures est considérablement simplifiée si chacun des membres d'un groupe dispose d'une vision cohérente de la sémantique du groupe (notamment l'ordre dans lesquels les messages sont délivrés). Ainsi, le groupe, en tant qu'abstraction, contient la sémantique de la communication entre ses membres et peut donc être considéré comme une interaction.

Parmi toutes les approches offrant une notion de groupes d'objets [VKC+99], on peut notamment citer Isis [BSS91, Bir93], Horus [GBC+93, RBM96], Psync/Consul [MPS93], GARF [GGM93], Newtop [EMS95], Transis [DM96, ABC+96], Totem [MMSA+96], OGS [FGW00].

Un groupe est constitué d'un ensemble de processus distribués. Un processus membre d'un groupe communique avec les autres membres uniquement par diffusion d'un message au groupe tout entier. Ainsi, lorsqu'un message est envoyé au groupe, tous ses membres le reçoivent. De ce fait, le support des interactions par des groupes ne permet d'exprimer que des relations pour lesquelles la réception d'un message implique la diffusion d'un même message à un ensemble d'objets. Ceci peut limiter considérablement le domaine d'expression des interactions par ces approches.

Les interactions définies à l'aide de groupes d'objets sont parfaitement adaptées pour exprimer la replication d'un objet. Dans ce cas, toutes les approches offrent un support transparent à l'exécution de ces interactions. En effet, l'objet client s'adresse à l'abstraction représentant le groupe comme il s'adresserait à l'objet serveur. Le système distribué se chargeant de diffuser le message à l'ensemble des membres du groupe.

L'utilisation de groupes d'objets pour décrire des interactions dont le domaine d'application ne concerne pas les systèmes de tolérance aux fautes est beaucoup plus délicate. En effet, dans ce cas, lorsqu'un message est envoyé au groupe, il est transmis à tous ses membres qui doivent décrire la sémantique qui leur est associé vis-à-vis de ce message. Ainsi, la sémantique de l'interaction est « distribuée » entre tous ses participants.

EXEMPLE. — La figure 2.11 décrit, en OGS, une calculatrice distribuée qui est répliquée. La classe décrivant la calculatrice hérite de la classe `mGroupAdmin::Groupable`. De ce fait, les instances de cette classe peuvent appartenir à des groupes d'objets. Le programme principal du serveur crée les objets de l'application (ici des calculatrices), crée un administrateur de groupe et enregistre les calculatrices dans ce groupe (via l'administrateur de groupe).
Le client, de son côte, récupère une référence sur le groupe (une interface de type`Calculator`) puis « dialogue » avec le groupe par le biais de cet objet. Il n'y a donc aucune différence, côté client, entre un objet simple et un objet dans un groupe (seule l'obtention de la référence est légèrement différente).

Lorsque l'envoi d'un message à l'un des membres d'un groupe est automatiquement (et de façon transparente) transmise à tous les autres membres du groupe (cas de OGS ou GARF), l'exécution de l'interaction décrite par le groupe est transparente pour l'objet émetteur du message. Par contre, si le message doit être envoyé à une entité encapsulant les membres du groupe, l'exécution de l'interaction ne sera pas transparente puisque l'objet émetteur devra explicitement appeler cette entité en lieu et place de l'objet membre du groupe.

```
// Server side
int main (int argc, char *argv[])
{
  // ORB Init
  CORBA::ORB_var orb = CORBA::ORB_init (...);
  CORBA::BOA_var boa = orb->BOA_init (...);

  // Create server application objects
  Calculator_var calc1 = new Calculator_i ();
  Calculator_var calc2 = new Calculator_i ();
  Calculator_var calc3 = new Calculator_i ();
  ...

  // Create a group administrator
  GroupAdministratorFactory_var gaf = ...;
  GroupAdministrator_var ga = gaf->create (...);

  // Join the group
  ga->join_group (calc1);
  ga->join_group (calc2);
  ga->join_group (calc3);
  ...

  // Wait for messages
  boa->impl_is_ready ();
  return 0;
}
```

```
// Client side
int main (int argc, char *argv[])
{
  // ORB Init
  ...

  // Get a reference to the interface
  // definition of the server
  CORBA::InterfaceDef_var intf =
  CORBA::InterfaceDef::_narrow
      (ir->lookup ("Calculator"));

  // Create a typed group accessor
  GroupAccessorFactory_var gaf = ...;
  CORBA::Calculator_var calc =
    CORBA::Calculator::_narrow
      (gaf->create_typed (..., intf));

  // Invoke replicated server
  calc->add (7);
  ...

  return 0;
}
```

FIG. 2.11 – *Exemple d'interaction en OGS*

Il est à noter que toutes ces approches permettent l'ajout et la suppression dynamique d'un membre dans un groupe ainsi que l'instanciation dynamique des groupes d'objets. Cependant, elles n'offrent qu'un ensemble fixe et non modifiable de sémantiques de communication (en général distribution non ordonnée, partiellement ordonnée et entièrement ordonnée des messages au sein du groupe). De plus, certaines approches, dont notamment Consul et Transis, ne permettent pas à un objet d'appartenir à plusieurs groupes (et donc interactions) simultanément.

En fait, toutes ces approches sont parfaitement adaptées à la définition d'interactions décrivant les concepts de tolérance aux fautes. L'intérêt apporté par ces approches pour le support des interactions pour un domaine autre que celui de la tolérance aux fautes est beaucoup moins tangible, voire négligeable.

2.4.4 Synthèse partielle

Le tableau 2.4 présente une synthèse du support des interactions par les approches étudiées dans ce paragraphe les plus représentatives.

Critères														
Abstraction des interactions				Intégration dans le modèle à objets				Expressions des interactions				Aspects dynamiques		
C1.1	C1.2	C1.3	C1.4	C2.1	C2.2	C2.3	C2.4	C3.1	C3.2	C3.3	C3.4	C4.1	C4.2	C4.3
MALEVA [Lhu98]														
✗	✗	✗	✓	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗
Jonathan [DTH+98]														
✘	✗	✓	✗	✘	✓	✓	✓	✓	✗	✗	✗	✘	✗	✗
Tj [McA96]														
✓	✓	✓	✓	✓	✓	✓	✘	✓	✗	✘	✓	✘	✗	✗
OGS [FGS98]														
✘	✗	✓	✘	✘	✓	✓	✓	✓	✗	✘	✗	✘	✗	✘

✓ : supporté, ✗ : non supporté, ✘ : supporté sous certaines conditions

TAB. 2.4 – *Synthèse du support des interactions par le biais d'autres mécanismes*

2.5 Synthèse

Dans ce paragraphe, nous récapitulons, de manière synthétique, les critères définis tout au long de ce chapitre. Nous les avons classés en 4 groupes : l'abstraction des interactions, l'intégration des interactions dans le modèle à objets, l'expression des interactions et le support des aspects dynamiques des interactions. Nous indiquons, pour chacun d'entre eux, ce que leur absence de prise en compte implique au niveau du support des interactions ou, inversement, les apports de cette prise en compte.

2.5.1 Abstraction des interactions

De cette étude, nous avons déterminé 4 critères concernant l'abstraction des interactions vis-à-vis des objets interagissants :

Critère C1.1 : Interaction = entité de première classe
Les interaction doivent disposer d'un niveau élevé d'abstraction et d'un très faible couplage vis-à-vis des objets interagissants.

Critère C1.2 : Héritage des interactions
La description des interactions doit pouvoir être réalisée de manière incrémentale, par raffinement, spécialisation et extension.

Critère C1.3 : Indépendance vis-à-vis des langages de programmation
La description des interactions doit être indépendante de tout modèle de programmation et de tout langage de programmation donné.

Critère C1.4 : Description formelle
La définition des interactions doit être réalisée de manière formelle (par description formelle des objets interagissants).

Nous avons observé que plus les interactions ont un niveau d'abstraction élevé, plus le couplage objets / interactions est faible. Or nous avons constaté qu'un couplage faible favorise la réutilisation, la maintenance et l'évolution des interactions et des objets interagissants. Cette abstraction se traduit, dans la plupart des cas (FLO [Duc97a], Synchronizers [AFP+93], OLAN [BR96] notamment), par une description des interactions dans une entité (« propriétaire » ou de première classe). En effet, cette entité favorise une description formelle de la sémantique des interactions ainsi que la définition d'un concept d'héritage entre les interactions [Duc97a]. Ce concept d'héritage ouvre la voie au raffinement des interactions et à la construction d'interactions au comportement complexe à partir d'interactions plus simples.

Cependant, bien que le concept des interactions soit indépendant de tout modèle de programmation, les approches étudiées n'assurent un support des interactions que pour un langage à objets donné ou un environnement de programmation donné. Ceci est la conséquence d'une abstraction insuffisante du concept d'interactions et de l'utilisation de concepts spécifiques aux langages à objets cibles.

La description formelle des interactions apportée par l'abstraction des interactions peut être comparée au concept de classe du modèle à objets. En effet, ces descriptions formelles peuvent être « instanciées » sur différents ensemble d'objets. Ceci est d'autant plus vrai que les entités décrivant les interactions disposent des propriétés des entités de première classe (on rappelle que seules quelques approches décrivent les interactions sous la forme de réelles entités de première classe). Nous appellerons *schémas d'interactions* de telles descriptions formelles des interactions.

2.5.2 Intégration dans le modèle à objets

De cette étude, nous avons déterminé 4 critères concernant l'intégration du support des interactions dans le modèle à objets :

Critère C2.1 : Transparence de l'exécution des interactions
Le support des interactions doit être transparent. Ceci implique que les objets interagissants n'ont pas besoin d'être préparé pour interagir et que l'exécution des interactions est entièrement gérée par le système.

Critère C2.2 : Respect des paradigmes du modèle sous-jacent
Les interactions doivent respecter les paradigmes du modèle sous-jacent et en particulier l'encapsulation.

Critère C2.3 : Utilisation des concepts de communication du modèle sous-jacent
Les interactions ne doivent pas définir de nouveaux concepts de communication mais, au contraire, utiliser ceux du modèle sous-jacent.

Critère C2.4 : Conservation des performances initiales
Le support des interactions ne doit pas dégrader sensiblement les performances globales du système.

À part quelques exceptions (Linda [Kie96], Dynamic Composite Adapter Pattern [SML99]), la présence d'une interaction est transparente pour les objets sur lesquels elle s'applique : un objet s'adresse aux autres objets de façon normale sans se préoccuper des interactions qui lui sont associées. Les interac-

tions sont gérées soit par le système lui-même [BR96, AB98], soit par contrôle (par capture) des envois ou des réceptions de messages [Duc97a], soit par transformation du code de l'application [LLM99].

Cependant, dans les approches nécessitant l'appel explicite de l'interaction, tel que le *framework* de [SML99] qui implique de réaliser un transtypage explicite pour appeler une interaction, l'ajout d'une interaction dans du code existant nécessite de repérer dans ce code tous les appels à l'objet interagissant et à effectuer à la main l'appel à l'interaction, avec tous les problèmes que cela comporte (oubli d'un appel par exemple). Ainsi, dans les approches ou les interactions sont transparentes, c'est le système (grâce aux mécanismes qu'il met en œuvre) qui gère la nouvelle interaction de manière automatique et, dans les autres cas, le support des interactions vient se greffer sur le modèle sous-jacent.

En fait, les approches dont la transparence du support des interactions n'est pas assurée définissent de nouveaux mécanismes de communication, tels que les flots d'événements (Java Beans [SUN97], CCM [OMG99]), pour gérer les interactions. Cette solution oblige le programmeur à repenser en termes d'interactions son application mais de manière non naturelle vis-à-vis des concepts de communication proposés par le modèle sous-jacent (envoi de messages notamment). En réalité, ces approches ne s'intègrent pas dans un modèle mais définissent un nouveau modèle de programmation.

2.5.3 Expression des interactions

De cette étude, nous avons déterminé 4 critères concernant le pouvoir d'expression apporté par les approches pour décrire les interactions :

Critère C3.1 : Interactions n-aires
Les interactions doivent permettre l'expression de la sémantique de la communication entre un ensemble quelconque d'objets.

Critère C3.2 : Aucune discrimination entre les objets au sujet du support des interactions
Le support des interactions doit être le même pour tous les objets, qu'ils soient distribués ou non.

Critère C3.3 : Décentralisation du support des interactions
La gestion des interactions (et notamment leur exécution) ne doit pas être basée sur une architecture centralisée mais, au contraire, sur une architecture « répartie ».

Critère C3.4 : Extension des sémantiques de communication
L'ensemble des sémantiques des communication utilisées pour décrire les interactions doit être extensible et modifiable.

Le pouvoir d'expression des interactions détermine, de manière significative, le domaine d'application des interactions. Mis à part dans FLO et dans l'approche *Dynamic Composite Adapter Pattern*, les interactions n'ont pas le même statut que les autres objets du système. En fait, les interactions ne sont pas considérées comme étant l'un des participants à part entière de l'interaction. Ceci est une conséquence indirecte d'un niveau trop faible de l'abstraction des interactions.

De plus, pour les approches offrant un support du distribué, les interactions ne peuvent être, soit définies que sur des objets distants, soit déclenchées que si les objets interagissants sont situés sur des sites distants. Nous avons par conséquent une discrimination des objets entre eux vis-à-vis du support des interactions, ce qui restreint les domaines d'utilisation des interactions et peut, dans certains cas, introduire des problèmes de cohérence.

Un autre élément de restriction est le support des interactions par une entité centrale. Ceci implique en effet que la sémantique de toutes les interactions est décrite dans une unique entité ayant en charge l'exécution de toutes les interactions. Dans un environnement distribué, cette entité peut devenir un goulot d'étranglement au niveau des ressources réseau et être un maillon faible en cas de panne réseau.

2.5.4 Aspects dynamiques

De cette étude, nous avons déterminé 3 critères concernant les aspects dynamiques associés aux interactions :

Critère C4.1 : Dynamicité de la définition des interactions
Les interactions doivent pouvoir être définies et déclarées dynamiquement lors de l'exécution de l'application sans nécessiter l'arrêt de cette dernière.

Critère C4.2 : Configuration dynamique
Les interactions doivent pouvoir être configurées (ajoutées, supprimées ou modifiées) dynamiquement.

Critère C4.3 : Dépôt des définitions formelles des interactions

Les définitions formelles des interactions doivent pouvoir être consultées, en vue de modification, d'ajout ou de suppression, lors de l'exécution du système.

De toutes les approches étudiées, seule Conic offre une gestion dynamique des interactions digne de ce nom. Or cet aspect est, à notre avis, primordial dans le cadre d'un environnement distribué.

Malheureusement, la plupart des autres approches ignorent purement et simplement l'aspect dynamique du support des interactions. Cependant, quelques unes autorisent la déclaration dynamique des interactions, permettant ainsi, de dissocier la création des interactions de celle des objets interagissants. En général, cette propriété est une conséquence de la description des interactions sous la forme d'une entité de première classe. Malgré ceci, ces approches ne permettent pas, sauf dans des environnements disposant de propriétés spécifiques (telle que la notion de chargement dynamique de classes) la définition de nouvelles interactions.

En fait, cette absence de support dynamique des interactions est le résultat de l'absence à l'exécution d'une représentation des descriptions formelles des interactions. La seule approche disposant d'une telle représentation est Conic. Elle « stocke » la description formelle des interactions dans un dépôt (*repository*) basé sur des concepts très similaires à ceux utilisés par le dépôt d'interfaces de CORBA.

2.6 Conclusion

Dans ce chapitre, nous avons étudié différentes approches qui introduisent le concept d'interaction dans les langages orientés objets. Nous avons extrait de ces approches un ensemble de critères. La prise en compte de ceux-ci nous semble primordiale afin d'offrir un support « idéal » aux interactions. En guise de conclusion, nous présentons un tableau récapitulatif (tableau 2.5 page ci-contre) dans lequel est indiqué, pour chacune des principales approches sur lesquelles a porté notre étude, quels sont les critères supportés.

Le chapitre suivant « instancie » chacun des critères dans le contexte de notre travail, à savoir un environnement compilé, fortement typé et distribué (C++ / Java et CORBA / RMI). Nous proposons ainsi un premier cadre pour le support des interactions dans le modèle à objets.

Critères														
Abstraction des interactions				Intégration dans le modèle à objets				Expressions des interactions				Aspects dynamiques		
C1.1	C1.2	C1.3	C1.4	C2.1	C2.2	C2.3	C2.4	C3.1	C3.2	C3.3	C3.4	C4.1	C4.2	C4.3
Modèle à objets [INR98]														
✗	✗	✗	✗	✗	✗	✗	-	✗	✗	✗	✗	✗	✗	✗
Modèle à composants [SML99]														
✓	✗	✗	✗	✓	✓	-	-	✗	✗	✗	✗	✗	✗	✗
Composition Filters [ABV92]														
✗	✗	✗	✗	✓	✗	✓	-	✓	✗	✓	✗	✗	✗	✗
Java Beans [Eng97]														
✓	✓	✗	✓	✗	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗
Synchronizers [FA93]														
✓	✓	✗	✓	✓	✓	✓	-	✓	✗	✓	✗	✗	✗	✗
FLO [Duc97b]														
✓	✓	✗	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✗	✗
Dynamic Composite Adapter Pattern [SML99]														
✓	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✗	✗	✗
Aspectual Components [LLM99]														
✓	✓	✓	✓	✓	✓	✓	✗	✓	✗	✓	✓	✗	✗	✗
JATLite [Jeo98]														
✓	✗	✗	✓	✗	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗
Darwin [MK96]														
✓	✓	✓	✓	✓	✓	✓	-	✓	✗	✓	✓	✗	✓	✗
OLAN [BR96]														
✓	✓	✓	✓	✓	✓	✓	-	✓	✗	✓	✗	✗	✗	✗
RAPIDE [Bry94]														
✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✗	✗
Conic [MKS89]														
✓	✓	✗	✓	✓	✓	✓	-	✓	✗	✓	✗	✓	✓	✓
COOL [BF95]														
✓	✗	✗	✗	✓	✓	✓	-	✓	✗	✓	✗	✗	✗	✗
MAF [BPC+95]														
✗	✗	✗	✓	✓	✓	✓	-	✓	✗	✓	✗	✗	✗	✗
Brainstorm/J [ZA00]														
✗	✗	✗	✓	✓	✓	✓	-	✓	✗	✓	✓	✗	✗	✗
CODA [McA95]														
✓	✓	✓	✓	✓	✗	✓	✗	✓	✗	✓	✗	✗	✗	✗
MALEVA [Lhu98]														
✗	✗	✗	✓	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗
Jonathan [DTH+98]														
✗	✗	✓	✗	✗	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗
Tj [McA96]														
✓	✓	✓	✓	✓	✓	✓	✗	✓	✗	✗	✓	✗	✗	✗
OGS [FGS98]														
✗	✗	✓	✗	✗	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗
C1.1	C1.2	C1.3	C1.4	C2.1	C2.2	C2.3	C2.4	C3.1	C3.2	C3.3	C3.4	C4.1	C4.2	C4.3
Abstraction des interactions				Intégration dans le modèle à objets				Expressions des interactions				Aspects dynamiques		

TAB. 2.5 – *Tableau de synthèse des principales approches étudiées supportant les interactions*

Chapitre 3

Intégration dans un environnement compilé, fortement typé et distribué

Ce chapitre est la suite logique du chapitre précédent. Dans ce dernier, nous avons déterminé un ensemble de critères permettant de caractériser le support des interactions de travaux existants. Cependant, les valeurs de ces critères sont souvent très dépendantes du modèle de programmation et de l'environnement de travail. Or, comme nous l'avons indiqué dans l'introduction, notre environnement de travail est un environnement compilé, fortement typé et distribué (C++ / Java avec CORBA / RMI). Nous présentons maintenant le support des interactions dans cet environnement. Au vu de cette étude, nous énonçons les valeurs pour chacun des critères et déterminons les caractéristiques essentielles que doit posséder, à notre avis, une intégration du support des interactions dans un environnement de type C++ / CORBA. Ces caractéristiques seront prises en compte dans notre description du modèle aux chapitres 5 et 6.

Le plan de ce chapitre est le suivant : nous proposons, pour chacun des mécanismes offerts par CORBA, une mise en œuvre possible du support des interactions et montrons leurs limites vis-à-vis des critères. Nous terminons par une présentation informelle de notre modèle, le modèle à interactions distribuées.

3.1 Support des interactions dans le modèle OMA de CORBA

L'Object Management Group (OMG) est un consortium, créé en 1989, d'industriels, de constructeurs et d'éditeurs informatiques. Il se compose actuellement de plus de 1000 membres. Il a notamment pour but de définir des spécifications à partir des technologies existantes proposées par ses membres. Ces spécifications concernent la réutilisation, la portabilité ainsi que l'interopérabilité de systèmes à objets. Elles sont soumises au consortium après appel à propositions, les *Request For Proposals* (RFP), sur un sujet technologique précis afin de l'intégrer à son architecture et sont adoptées après un processus de vote des membres de l'OMG.

Parmi les principaux résultats de l'OMG, il y a la définition de l'architecture *Object Management Architecture* [OMA97] (OMA) et de CORBA [OMG98]. CORBA est la spécification d'un intermédiaire, un bus logiciel, entre des clients et des serveurs dans un système distribué. CORBA tente de résoudre des problèmes inhérents à la programmation distribuée tels que le support de l'hétérogénéité des plates-formes, l'interopérabilité des systèmes en ajoutant un *Object Request Broker* (ORB) qui joue le rôle de médiateur entre les clients et les serveurs.

Après avoir très brièvement décrit l'architecture du modèle CORBA, nous décrivons quatre mécanismes permettant de décrire les interactions dans CORBA. Nous terminons cette présentation par une discussion sur le support des interactions apporté par ces différents mécanismes.

3.1.1 Architecture OMA

L'architecture OMA fournit l'infrastructure conceptuelle sur laquelle sont basées les spécifications de l'OMG concernant le bus logiciel (*middleware*). Elle est l'une des composantes [1] de la vision générale de l'OMG nommée *Model Driven Architecture* (MDA) [MDA01].

L'architecture OMA se compose de cinq composants indépendants les uns des autres, présentés dans la figure 3.1 et définis très succinctement ci-après.

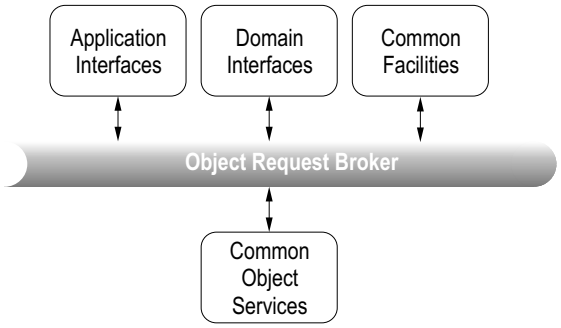


FIG. 3.1 – *Architecture OMA*

Object Request Broker (ORB)

L'ORB est le composant central de l'architecture OMA. Il peut être vu comme la moelle épinière de l'architecture. C'est en effet lui qui est responsable de la communication entre tous les objets du système. Il assure la transparence de la communication lorsque les objets en communication sont distants ou sur des systèmes hétérogènes.

La fonction de l'ORB est analogue à celle d'un bus matériel, dans le sens où il fournit un canal de communication entre les objets. Comme un tel bus, il autorise la connexion et la déconnexion dynamique des objets (qu'ils soient clients ou serveurs). Les autres composantes de l'architecture OMA utilisent l'ORB et décrivent des interfaces de spécification plus évoluées.

Common Object Services (COS)

Ces interfaces décrivent un ensemble prédéfini de fonctions génériques permettant la gestion des objets (création, contrôle d'accès, etc.) et des communications (transactions, notification d'événements, etc.). Les *COS* [OMG97a] ont été définis afin d'aider le développeur d'applications. En effet, il peut utiliser ces services, dont l'interface est normalisée, pour créer ses propres composants.

EXEMPLE.— Parmi les services définis, on peut noter le service de nommage des objets (pour localiser les objets), le service de notification d'événements, le service de persistance et le service de gestion du cycle de vie des objets (pour contrôler la création et la destruction des objets).

Les *COS* fournissent un ensemble de services, sous la forme de librairies d'exécution ou de serveurs, qui supportent la création d'applications distribuées orientés objet dans l'environnement CORBA. Un *COS* est générique (c'est-à-dire indépendant du domaine de l'application), décrit en *Interface Definition Language* (IDL) [Lam87], et se construit au-dessus de l'ORB.

Common Facilities et Domain Interfaces

Les CORBA *Facilities* proposent un ensemble d'abstractions d'intérêt général simplifiant le développement des applications telles que la gestion de documents composites, l'accès aux bases de données, les services d'impression, etc. Ce sont des interfaces indépendantes du domaine de l'application. Elles sont plutôt rares.

Ces interfaces peuvent être classées en quatre groupes : l'interface utilisateur, la gestion de l'information, l'administration du système et la gestion des tâches. Les *Domain Interfaces* sont similaires aux CORBA *Facilities* mais elles dépendent du domaine de l'application. Elles sont de plus en plus nombreuses.

Application Interfaces

Ces interfaces fournissent un ensemble qui réalisent des tâches spécifiques pour une application donnée. Ces interfaces peuvent être construites à partir d'autres interfaces plus basiques, certaines spécifiques à l'application et d'autres provenant des *Common Facilities* ou des *COS*. Ces interfaces ne sont, et ne seront jamais, normalisées par l'OMG.

EXEMPLE.— Une application de traitement de textes peut être construite en utilisant des interfaces particulières à l'application telles que la création et la gestion d'un document, et des interfaces provenant des CORBA *Facilities* pour gérer l'impression et la sauvegarde d'un document.

1. Les autres composantes sont *Meta-Object Facility* (MOF) [MOF97] et *Unified Model Language* (UML) [BJR97] et *Common Warehouse Metamodel* (CWM) [CWM01].

3.1.2 Expression des interactions avec les mécanismes de CORBA

CORBA applique les principaux concepts objets au modèle OMA mais offre des mécanismes de communications entre objets distants (le modèle à objets ne propose que l'envoi de messages). Le mécanisme principal, et par défaut, est le mécanisme *Static Invocation Interface* (SII) qui offre la même sémantique que l'envoi de messages dans le modèle à objets. Le second mécanisme est le *Dynamic Invocation Interface* (DII) qui permet de construire, lors de l'exécution, une requête CORBA. Chacun de ces deux mécanismes a des avantages et des inconvénients vis-à-vis du support des interactions.

L'architecture d'OMA fournit une présentation organisée des concepts et technologies objets. Il sépare les *clients* des *serveurs* en encapsulant de manière parfaitement définie leurs interfaces. Ainsi un *client* ne connaît d'un *serveur* que sa structure logique, décrite par son interface en IDL. La sémantique de l'objet étant définie par sa mise en œuvre.

Ce modèle présente des concepts qui sont nécessaires aux clients, tels que le concept d'objets (des entités encapsulées fournissant un ensemble de services aux clients), de requêtes (délivrées lorsqu'un client demande un service à un objet) et d'opérations (services pouvant être invoqués, identifiés par un nom et possédant une signature), de typage et de signatures (spécification des paramètres et du résultat).

Expression des interactions avec le mécanisme SII

Le langage IDL permet uniquement de décrire les interfaces des objets pouvant être accessibles au travers de l'ORB. Il ne propose aucun support explicite des interactions. Celles-ci devront être mises en œuvre dans le code des objets à l'aide du mécanisme SII. Ainsi, le code de l'interaction est mélangé à celui de l'objet.

La nécessité de décrire les interactions dans le code des objets signifie qu'elles doivent être définies lors de la phase de conception. Il n'est donc, par conséquent, pas possible de rajouter une interaction en cours d'exécution de l'application (à moins d'arrêter l'application, de modifier son code source et de la recompiler). Cependant les participants de l'interaction peuvent être déterminés lors de l'exécution.

EXEMPLE.— La figure 3.2 montre comment écrire une interaction en CORBA. Cet exemple (compilé à l'aide de l'ORB ORBACUS [OOC98]) présente le comportement de déplacement d'un objet graphique (provenant d'un éditeur graphique vectoriel) en fonction de son appartenance, ou non, à un groupe d'objets.

```
interface Figure
{
  void move (in long dx, in long dy);
  boolean isGroupedWith (in Object obj);
  ...
};

interface Point : Figure { };
```

```
class FigImpl : virtual public Figure_skel
{
  public:
    void move (CORBA_Long, CORBA_Long);
    ...

  protected:
    static List <Fig_var> figList;
    static Display_var display;
};

class PointImpl : virtual public Point_skel,
                  virtual public FigImpl
{
  public:
    void move (CORBA_Long dx,
               CORBA_Long dy);
    ...

  private:
    int x, y;
}
```

```
void FigImpl::move (CORBA_Long dx, CORBA_Long dy)
{
  ListIterator iter (figList);

  while (iter.hasMoreElement ())
  {
    if (iter.getElement ()->isGroupedWith (this))
      iter.getElement ()->move (dx, dy);

    iter.nextElement ();
  }

  display->update ();
}

void PointImpl::move (CORBA_Long dx, CORBA_Long dy)
{
  if (!alreadyInMove)
  {
    alreadyInMove = true;

    x += dx; y += dy;

    FigureImpl::move (dx, dy);

    alreadyInMove = false;
  }
}
```

FIG. 3.2 – *Exemple d'interaction en* CORBA *avec le mécanisme SII*

On peut voir que le code de l'interaction (code sur fond gris) est mélangé à celui de l'objet, qu'une partie non négligeable du code (code en italique) de l'interaction concerne la détermination de l'état d'activité[2] de l'interaction (fonction de son appartenance ou non à un groupe d'objets). Le code en italique détermine, pour chaque objet du dessin, s'il appartient au

2. L'état d'activité d'une interaction correspond à déterminer si l'interaction est active, c'est-à-dire présente dans le système, et donc doit être exécutée ou si elle est inactive, c'est-à-dire « absente » du système.

même groupe et, lorsque c'est le cas, déplace cet objet (exécute une partie de l'interaction). Une fois que tous les objets sont déplacés l'affichage est mis à jour.

Expression des interactions avec le mécanisme DII

CORBA offre un mécanisme permettant de construire, lors de l'exécution, une requête CORBA puis de l'envoyer sur le bus ORB. Ce mécanisme est appelé DII. La communication avec le DII peut être synchrone ou asynchrone. Ce mécanisme est d'ailleurs le seul moyen pour effectuer des requêtes asynchrones.

La construction lors de l'exécution de la requête permet la définition de schémas d'interactions (et non plus d'interaction comme cela est le cas avec le mécanisme SII). Un schéma d'interactions peut être considéré comme un schéma de conception (*design pattern* [GHJ+95]). Cette construction dynamique permet de définir des schémas d'interactions génériques, aussi bien sur les objets participants que sur les méthodes à exécuter sur ces objets.

De plus, l'interaction peut être définie dans un objet spécifique (qui décrit le schéma d'interactions) et n'est donc plus mélangée au code des objets participants. Cependant, cela implique d'appeler explicitement la méthode décrivant la sémantique de l'interaction. Ainsi, l'exécution de l'interaction n'est pas transparente. De plus, tout comme pour le mécanisme SII la définition à l'exécution de nouveaux schémas d'interaction n'est pas possible.

EXEMPLE. — La figure 3.3 présente, pour le même exemple que celui de la figure 3.2, une autre interaction définie, cette fois-ci, à l'aide du mécanisme DII. Elle définit un schéma d'interactions (code sur fond gris) qui exécute, de manière asynchrone, une même méthode sur deux objets différents.

Dans cet exemple, l'interaction est définie dans un objet spécifique (qui décrit le schéma d'interactions) et n'est donc plus mélangée au code des objets participants (comme cela est le cas dans l'exemple décrivant le mécanisme SII). La requête de communication est construite par cet objet lors de l'appel de sa méthode `apply` dont le rôle est d'exécuter la sémantique de l'interaction. Le code en italique représente l'appel explicite de l'interaction.

```
class Interaction
{
  public:
    void apply (CORBA_Long dx,
                CORBA_Long dy,
                const char *method,
                CORBA_Object_var obj1,
                CORBA_Object_var obj2);
}

class LigneImpl : virtual public Ligne_skel,
                  virtual public FigImpl
{
  public:
    void move (CORBA_Long, CORBA_Long);
    ...

  protected:
    Point_var pt1, pt2;
};

void LigneImpl::move (CORBA_Long dx,
                      CORBA_Long dy)
{
  Interaction interaction ();

  interaction.apply (dx, dy, "move", pt1, pt2);

  FigImpl::move (dx, dy);
}
```

```
void Interaction::apply ( ... )
{
  CORBA_Request_var req;
  CORBA_NVList_ptr nvlist;
  CORBA_Any *arg;

  orb->create_list (2, nvlist);

  arg = nvlist->add (CORBA_ARG_IN)->value ();
  arg <<= dx;

  arg = nvlist->add (CORBA_ARG_IN)->value ();
  arg <<= dy;

  obj1->_create_request (0, method, nvlist,
                         0, 0, 0, req, 0);
  req->send_one_way ();

  obj2->_create_request (0, method, nvlist,
                         0, 0, 0, req, 0);
  req->send_one_way ();
}
```

FIG. 3.3 – *Exemple d'interaction en* CORBA *avec le mécanisme DII*

3.1.3 Expression des interactions à l'aide des Event et Relationship Services

Les mécanismes standards de CORBA que sont SII et DII n'offrent donc qu'un support très limité à l'expression explicite et à la gestion des interactions. Cependant, deux services permettant une meilleure abstraction des interactions ont été définis au dessus de la norme CORBA 2.0. Il s'agit des services *Event Service* [OMG97b] et *Relationship Service* [OMG97c]. Nous montrons dans ce paragraphe comment mettre en œuvre ces deux services et comment ils sont complémentaires pour pouvoir décrire et gérer les interactions et quelles sont les limites de cette approche.

Présentation du Event Service

Une requête standard (c'est-à-dire avec le mécanisme SII) en CORBA résulte d'une exécution synchrone d'une opération par un objet. Si cette opération définit des paramètres ou des valeurs de retour, les données sont transmises entre le client et le serveur. Une requête est destinée à un objet particulier. Ainsi pour que la requête réussisse, le client et le serveur doivent être disponibles, le client devant connaître le serveur. Si la requête échoue par une indisponibilité du serveur, le client reçoit une exception et doit prendre des mesures appropriées.

Le service COS *Event Service* [OMG97b] permet de découpler les communications entre les objets. Ainsi il définit deux rôles pour les objets : le rôle de *producteurs* (*supplier role*) et le rôle de *consommateurs* (*consumer role*). Les *producteurs* produisent des *données événementielles* (*event data*) tandis que les *consommateurs* traitent ces données. Les données événementielles sont transmises entre les producteurs et les consommateurs en utilisant les mécanismes standards de CORBA. Il est à noter que producteurs et consommateurs n'ont pas besoin de se connaître.

Pour ce faire, le Event Service définit la notion de *canal d'événements* (*event channel*). Il s'agit d'un objet qui permet à plusieurs producteurs de communiquer avec plusieurs consommateurs de façon asynchrone. Un canal d'événements est à la fois un producteur et un consommateur d'événements (figure 3.4).

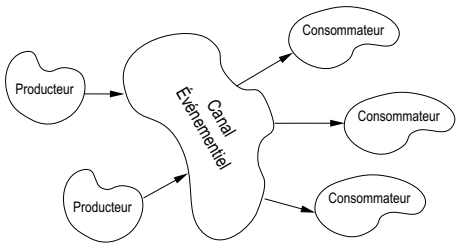


FIG. 3.4 – *Vue d'ensemble du Event Service*

Les canaux d'événements sont des objets CORBA standards et les communications avec un canal d'événements est accompli en utilisant les requêtes CORBA standards.

Présentation du Relationship Service

Les objets distribués sont fréquemment utilisés pour modéliser des entités du monde réel. Ainsi, ces objets distribués n'existent pas par eux-mêmes mais sont en relation avec d'autres objets. De telles relations peuvent être caractérisées par des attributs de type, de rôle au sein d'une relation, de degré, de cardinalité et par la sémantique d'une relation. Ce sont ces attributs que le Relationship Service gère.

Le type des relations entre entités contraint le type des entités de la relation. Par exemple, on peut définir une relation ***emploi*** qui est définie entre des ***personnes*** et des ***entreprises***. Ainsi une entreprise ne peut pas employer un singe car un singe n'est pas une personne. De plus, l'*emploi* est une relation distincte des autres relations qui peuvent exister entre une personne et une entreprise.

Une relation est définie par un ensemble de rôles que possède les entités. Dans la relation *emploi*, une entreprise joue le rôle de l'***employeur*** et une personne joue le rôle de l'***employé***. Une même entité peut avoir différents rôles dans différentes relations. Remarquez qu'une personne peut jouer le rôle du propriétaire dans une relation de propriété et le rôle de l'employé dans une relation d'emploi. Le nombre de rôles requis par une relation est défini par un attribut, le *degré*.

Le service COS *Relationship Service* [OMG97c] permet aux relations d'être explicitement représentées. Les entités sont représentées comme des objets CORBA. Le service définit deux nouveaux genres d'objets : les *relations* (*relationships*) et des *rôles* (*roles*). Un rôle représente un objet CORBA dans une relation. Une relation est créée en passant un ensemble de rôles à une fabrique de relations (*relationship factory*).

Support offert aux interactions

Le service COS *Relationship Service* permet d'expliciter le graphe des interactions. En effet, chaque participant d'une interaction, une relation pour le *Relationship Service*, possède un rôle. Il y a donc unification parfaite entre interactions et relations. Les mécanismes de parcours des relations mis à disposition par le *Relationship Service* permettent ainsi de parcourir le graphe des interactions.

Le service COS *Event Service* permet, quand à lui, de décrire une interaction sous la forme d'un canal d'événements que l'on peut comparer à la notion de connecteur dans le modèle à composants. Cependant, l'émission d'un message au travers d'un canal d'événements est explicite. Ainsi, l'exécution de l'interaction n'est pas transparente vis-à-vis des objets interagissants. Il est à noter que l'objet interagissant n'est pas obligé de connaître, lors de la phase de conception, le canal d'événements, et

donc l'interaction, sur lequel il émet les messages puisque ce canal est instancié indépendamment de l'objet interagissant.

EXEMPLE. — Dans l'exemple de la figure 3.5, concernant un éditeur graphique vectoriel, chaque objet composant le dessin transmet (*push*) à travers le canal d'événements (s'il est instancié) ses messages. Ce canal va ensuite traiter ces messages conformément à la sémantique de l'interaction.
Le code sur fond gris représente le code nécessaire pour permettre à l'objet de supporter les interactions. Le code en italique représente l'appel explicite de l'interaction. La fonction`connect_interaction` décrit l'instanciation de l'interaction.

```
class LigneImpl : virtual public Ligne_skel,
                  virtual public PushSupplier,
                  virtual public FigImpl
{
  public:
    void move (CORBA_Long, CORBA_Long);
    void connect_interaction (EventChannel_var);
    void disconnect_interaction ();

  protected:
    Point_var pt1, pt2;
    PushConsumer_var proxy_push_consumer;
};
```

```
void LigneImpl::move (CORBA_Long dx,
                      CORBA_Long dy)
{

  if (proxy_push_consumer)
    proxy_push_consumer->push ( ... );

  FigImpl::move (dx, dy);
}
```

```
void LigneImpl::connect_interaction (EventChannel_var ec)
{
  SupplierAdmin_var supplier = ec->for_consumers ();

  proxy_push_consumer = supplier->obtain_push_consumers ();
  proxy_push_consumer->connect_push_supplier (this);
}
```

FIG. 3.5 – *Exemple d'interaction à l'aide du service* CORBA *Event Service*

3.1.4 Aide à l'expression des interactions : langage de scripts et ORB réflexif

Langage de scripts

Ces dernières années est apparue la notion de scripts pour CORBA. CorbaScript [MGG96] (une mise en œuvre de la norme OMG IDLScript) est le langage de scripts pour CORBA : il permet d'exécuter des appels CORBA à l'exécution sous forme de fichiers batch ou en ligne de commandes (accès au bus CORBA, et donc aux objets CORBA, de manière interactive). CorbaScript est un véritable langage orienté objet (notion de polymorphisme, héritage, introspection, etc.). Il permet, en fait, de manipuler dynamiquement l'*Interface Repository* (IR) du bus CORBA.

Il peut donc être utilisé pour instancier dynamiquement des interactions à l'aide du Event Service (qui ne permet, lui, que des instanciations statiques ou définies à la compilation). Ainsi, il apporte la dynamicité de la gestion des interactions à CORBA et au Event Service. Cependant, il reste, malgré tout, tributaire de ce service. En effet, les objets doivent être conçus pour interagir en utilisant le Event Service : le script crée un canal d'événements et connecte les objets interagissants à ce canal.

Il est à noter que l'utilisation de CorbaScript sans le Event Service est également possible. Cependant le support des interactions est très fortement contraint. En effet, il devient très difficile de « poser » une interaction entre un ensemble d'objets qui sera déclenchée à chaque invocation d'un message. En fait, dans ce cas, il est uniquement possible de déclencher manuellement l'interaction.

Pour résumer, CorbaScript apporte l'instanciation et la définition dynamique des interactions aux autres mécanismes offrant un support des interactions dans CORBA. Cependant, l'exécution des interactions n'est pas transparente et doit être explicitement décrite par le programmeur. En fait, rendre cette exécution transparente implique une modification de la sémantique du mécanisme d'invocation distante.

ORB réflexif

Malheureusement, bien que l'architecture de CORBA soit modulaire (figure 3.1), le bus logiciel, l'ORB, reste, lui, un élément opaque et complètement fermé. Par conséquent, redéfinir la sémantique d'invocation distance implique « d'ouvrir » l'ORB. C'est justement ce qui a été fait par T. LEDOUX dans sa thèse [Led98]. En effet, il a défini un ORB réflexif (compatible avec CORBA 2.0) permettant, notamment, l'introspection et la modification des mécanismes d'exécution du bus logiciel. Ceci est réalisé en utilisant les techniques de réflexion dans le langage SmallTalk [Dug90].

L'utilisation d'un ORB réflexif ouvre des perspectives pour réaliser un support de l'exécution des interactions. En effet, cet ORB permet de simplifier la mise en œuvre des interactions et la possibilité de redéfinir la sémantique de l'envoi de messages à distance permet une exécution transparente des interactions.

Cependant, la présence d'un ORB réflexif ne suffit pas pour offrir un support complet aux interactions. En fait, les interactions doivent toujours être décrites à l'aide des concepts de base de CORBA (tel que, par exemple, les canaux d'événements). En fait, la réflexivité apportée par l'ORB sert simplement à masquer les détails de mise en œuvre des mécanismes internes à la gestion des interactions.

EXEMPLE. — L'exemple de la figure 3.6 reprend l'exemple précédent (éditeur graphique vectoriel). Les interactions sont définis à l'aide du *Event Service*. Il montre la définition et l'instanciation dynamique d'une interaction à l'aide de CorbaScript. On peut noter que, comme pour l'exemple de la figure 3.5, l'objet interagissant doit être préparé (code sur fond gris) pour pouvoir déclencher l'interaction.

```
class Ligne (Figure,
             PushSupplier,
             PushConsumer)
{
  proc __Ligne__ (self)
  {
    self.proxy_push_consumer = Void
  }

  proc move (self, dx, dy)
  {
    if (self.proxy_push_consumer != Void)
      self.proxy_push_consumer.push ( ... )

    Figure.move (self, dx, dy)
  }

  proc push (self, data)
  {
    Figure.move ( ... )
  }
}
```

```
proc instanciate_interaction (lignes)
{
  ec = EventChannel (CORBA.ORB.string_to_object (...))

  consummers = ec.for_consumers ()
  push_supplier = consummers.obtain_push_supplier ()

  suppliers = ec.for_suppliers ()
  push_consumer = suppliers.obtain_push_consumer ()

  for i in range (0,lignes.length - 1)
  {
    lignes [i].proxy_push_consumer = push_consumer
    push_consumer.connect_push_consumer (lignes [i])
    push_supplier.connect_push_supplier (lignes [i])
  }
}
```

FIG. 3.6 – *Exemple d'interaction à l'aide d'un langage de scripts* CORBA

3.1.5 Discussion : limites de ces approches

Les mécanismes de base de CORBA (SII et DII) n'offrent qu'un support plus que limité aux interactions. En réalité, CORBA applique les concepts et les paradigmes du modèle à objets à son architecture distribuée. Or il a été vu dans le chapitre précédent que ce modèle n'intégrait aucun support explicite des interactions. Il n'est donc pas surprenant que CORBA n'intègre pas, lui non plus, ce support.

Cependant, l'utilisation du mécanisme DII permet de définir des schémas d'interactions génériques qui pourront être instanciés sur des objets et des méthodes particulières lors de l'exécution. Ceci permet d'instancier dynamiquement de nouvelles interactions basées sur un schéma d'interactions. Cette solution est néanmoins très limitée car, lors de l'exécution, la définition de nouveaux schémas d'interactions n'est pas possible.

Néanmoins, la propriété de transparence des interactions n'est pas conservée lors de l'utilisation de ces schémas d'interactions puisque l'objet représentant le schéma d'interactions doit être explicitement construit et la méthode exécutant l'interaction explicitement appelée.

Heureusement, parmi les services définis au dessus de la norme 2.0 de CORBA, le *Event Service* et le *Relationship Service* offrent un support de bonne facture aux interactions. Ils permettent en effet de décrire les interactions sous la forme d'une entité distincte des objets interagissants, le canal d'événements (utilisation du *Event Service*). Ce canal découple l'objet initiateur de l'exécution d'une interaction des objets participants à l'interaction. De plus, le service *Relationship Service* permet le parcours du graphe des interactions présentes dans le système.

Il est à noter que la description des interactions grâce à ces services CORBA permet, lors de l'exécution, la découverte de nouvelles interactions non définies lors de la phase de conception. Ceci est possible car une interaction est représentée par un objet « canal d'événements » (*event channel*) du service CORBA *Event Service* qui est un objet CORBA défini par son IDL et donc offrant un « type » commun à toutes les interactions. Cependant, cela est entièrement à la charge du programmeur de l'application. En fait, tout repose sur la découverte dynamique de nouveaux serveurs CORBA et sur la définition d'un protocole de communication entre les serveurs et l'application. Ceci implique également l'utilisation de canaux d'événements non typés.

En fait, cette solution, combinant ces deux services CORBA, serait quasiment parfaite si le support des interactions était totalement transparent et pleinement dynamique. Il requiert en effet, pour qu'une interaction s'exécute, des appels explicites au canal d'événements représentant l'interaction. Par consé-

quent, il nécessite une préparation des objets initiateurs des interactions. De plus, un canal d'événements étant unidirectionnel il est nécessaire de disposer de deux canaux d'événements pour gérer un appel de méthode (un canal pour l'émission du message, l'autre pour récupérer la valeur de retour de l'envoi).

Les langages de scripts pour CORBA permettent de combler cette absence partielle de dynamicité tandis que l'utilisation d'un ORB réflexif permet de rendre transparent la gestion des interactions. L'OMG a normalisé les spécifications des langages de scripts pour CORBA [ISS00]. Ainsi, ceux-ci peuvent être utilisés sans soucis de compatibilité inter-ORB (interopérabilité). Ce n'est malheureusement pas le cas pour l'ORB réflexif. En effet, un seul ORB réflexif existe. Il s'agit d'OpenCorba [Led98], un ORB « ouvert » en Smalltalk. De ce fait, la définition d'un support transparent des interactions dans un système CORBA est fortement contraint (un seul langage, un seul bus logiciel, pas de normalisation auprès de l'OMG).

Le tableau 3.1 présente une synthèse du support des interactions par les différents mécanismes offerts par CORBA. Pour chaque mécanisme nous indiquons, dans ce tableau, si il supporte (symbole ✓), ne supporte pas (symbole ✗) ou supporte sous certaines conditions (symbole ✘) chacun des critères.

Critères														
Abstraction des interactions				Intégration dans le modèle à objets				Expressions des interactions				Aspects dynamiques		
C1.1	C1.2	C1.3	C1.4	C2.1	C2.2	C2.3	C2.4	C3.1	C3.2	C3.3	C3.4	C4.1	C4.2	C4.3
CORBA SII														
✗	✗	✓	✗	✗	✓	✓	-	✗	✗	✗	✗	✗	✓	✗
CORBA DII														
✘	✘	✓	✓	✗	✓	✓	-	✓	✗	✓	✗	✘	✗	✘
CORBA Event Service et RelationShip Servivce														
✓	✗	✓	✗	✘	✓	✘	✓	✓	✗	✓	✗	✗	✘	✗
CorbaScript et CORBA Event Service														
✓	✗	✓	✘	✘	✓	✘	✘	✓	✗	✓	✗	✓	✓	✗

TAB. 3.1 – *Synthèse du support des interactions dans le modèle* CORBA

3.2 Proposition d'une intégration dans le modèle C++ / CORBA

Un des objectifs de cette thèse est l'introduction du support des interactions dans des langages et des systèmes distribués existants. Parmi la multitude de couple *langage + système distribué* possibles, nous nous sommes restreints à ceux évoluant dans un environnement compilé et fortement typé, car ce sont ceux qui sont les plus utilisés dans le milieu industriel.

Nous donnons, dans ce paragraphe, pour chacun des différents critères définis dans le précédent chapitre, les caractéristiques de l'intégration des interactions qu'il implique. Ce sont ces caractéristiques qui seront prises en compte par notre modèle (décrit dans la partie II).

Nous rappelons les critères définis :

- Abstraction des interactions
  - C1.1 : Interaction = entité de première classe
  - C1.2 : Héritage des interactions
  - C1.3 : Indépendance vis-à-vis des langages de programmation
  - C1.4 : Description formelle
- Intégration dans le modèle à objets
  - C2.1 : Transparence de l'exécution des interactions
  - C2.2 : Respect des paradigmes du modèle sous-jacent
  - C2.3 : Utilisation des concepts de communication du modèle sous-jacent
  - C2.4 : Conservation des performances initiales

– Expression des interactions
  – C3.1 : Interactions n-aires
  – C3.2 : Aucune discrimination entre les objets au sujet du support des interactions
  – C3.3 : Décentralisation du support des interactions
  – C3.4 : Extension des sémantiques de communication
– Aspects dynamiques
  – C4.1 : Dynamicité de la définition des interactions
  – C4.2 : Configuration dynamique
  – C4.3 : Dépôt des définitions formelles des interactions

3.2.1 Abstraction des interactions

De l'étude bibliographique présenté dans le chapitre précédent, nous avons observé que plus le niveau d'abstraction est élevé et plus la réutilisation, l'extension et la maintenance des objets, comme des interactions, sont favorisées. La définition des interactions sous la forme d'une entité de première classe est la solution qui offre le meilleur niveau d'abstraction [Duc97b]. En effet, elle permet de spécifier clairement le comportement des interactions et de dissocier la définition et la déclaration des interactions de celles des objets interagissants. Par conséquent, **nous représentons les interactions par des objets** (afin de satisfaire au critère C1.1).

Le concept d'interaction est indépendant de tout modèle de programmation. Ainsi, à la manière des approches qui décrivent les interactions à l'aide d'un langage particulier (généralement un ADL), **nous proposons un langage conçu spécifiquement pour décrire les interactions de manière formelle** (critères C1.3 et C1.4). Nous appelons ce langage *Interaction Specification Language* (ISL).

La description formelle des interactions dans ce langage sera traduite sous la forme de classes (étendues) dans le langage applicatif. Nous appelons *schémas d'interactions* de telles classes. Ainsi, **nos interactions disposent de la notion de définition incrémentale** (critère C1.2), issue du mécanisme d'héritage du modèle à objets sous-jacent (par exemple, héritage multiple dans le cas de C++, simple dans celui de Java).

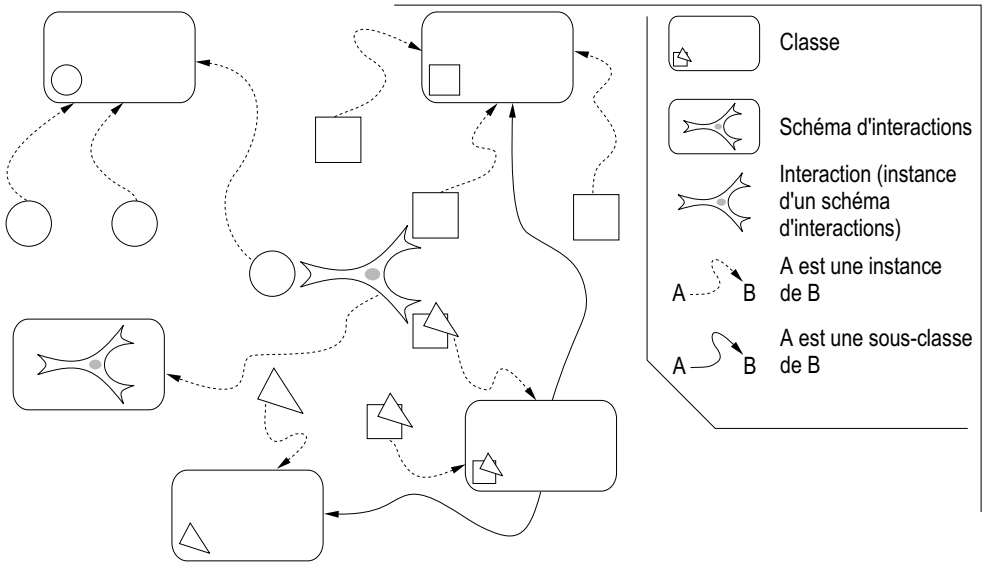


FIG. 3.7 – *Situation entre classes, instances, schémas d'interactions et interactions*

Par conséquent, les programmeurs définissent les classes, créent des instances de ces classes, *définissent* des schémas d'interactions et créent des *instances* de ces schémas d'interactions, les interactions, en déclarant les objets participants à ces interactions (figure 3.7). La définition et l'instanciation des schémas d'interactions est réalisée sans modifier les classes des objets participants aux interactions.

Ainsi les schémas d'interactions et les interactions sont dissociés des classes et des objets de l'application: la *définition* d'un schéma d'interactions n'est liée d'aucune manière à celle d'une classe de l'application et l'*instanciation* d'un schéma d'interactions n'est liée à aucune instanciation d'objet applicatif. Ceci permet d'instancier un schéma d'interactions bien après l'instanciation des objets qui vont y participer et de supprimer cette interaction bien avant la destruction des objets participants.

3.2.2 Intégration dans le modèle à objets

Notre objectif est d'offrir un support des interactions à un environnement de type C++ / CORBA, et ce avec une empreinte sur le système la plus faible possible (la vérification du critère C2.4 fait donc partie de nos objectifs). Ainsi, une interaction ne doit pas être invoquée explicitement par le programmeur mais, au contraire, être entièrement à la charge du système (critère C2.1). Par conséquent, **notre modèle doit intégrer un mécanisme d'exécution des interactions**.

Nous avons constaté que les approches mettant en œuvre de nouvelles sémantiques de communication pour gérer les interactions ne vérifiaient pas le critère de transparence. Le modèle à objets ne définit qu'une seule sémantique de communication, l'envoi de messages [Coi87]. De ce fait, **la sémantique des interactions doit être basée sur le contrôle de l'envoi de messages** pour les objets participants à une interaction (critère C2.3).

En total accord avec le critère C2.2, **une interaction doit respecter l'encapsulation des objets** qui sont ses participants. Ce respect de l'encapsulation est conforté par le fait que nous n'exprimons que des interactions entre objets. Par conséquent, la description de la sémantique d'une interaction ne doit être réalisée qu'à l'aide des méthodes définies dans l'interface des objets.

3.2.3 Expression des interactions

Nous avons vu dans le paragraphe 3.2.1 que les interactions sont décrites formellement à l'aide du langage ISL. Cette description est réalisée de **manière déclarative**, à l'aide d'**opérateurs réactifs**, chacun décrivant une sémantique de haut niveau (exécution synchrone, asynchrone, avec valeur de retour anticipée, etc.). Au vu du critère C3.4, **cet ensemble d'opérateurs doit être extensible**.

Une interaction ne doit être limitée ni sur son nombre d'objets participants, ni sur une discrimination des participants. Ainsi, une interaction peut être définie sur un nombre quelconque d'objets interagissants (critère C3.1). Étant un objet, une interaction peut, par conséquent, être l'un des participants d'une interaction. Plus précisément, une interaction est un participant à part entière pour elle même (vérification du critère C3.2).

Toujours afin de satisfaire au critère C3.2, une interaction peut être définie aussi bien entre des objets distribués qu'entre des objets non prévus pour être distribués. De ce fait, notre modèle à interactions distribuées va introduire le concept de référencement à tous les objets du système (qu'ils soient ou non prévus pour être distribués) afin que des interactions distribuées puissent être instanciées entre des objets non prévus pour être distribués. Ceci sera réalisé en étendant les mécanismes de référencement de l'architecture distribuée sous-jacente (figure 3.8).

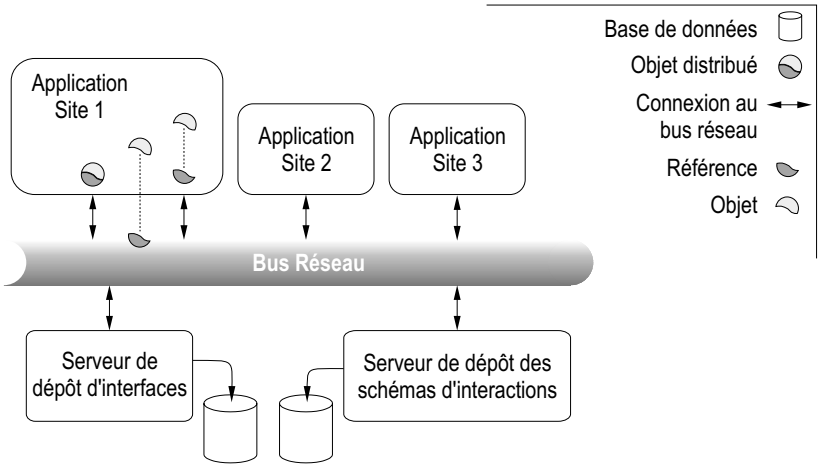


FIG. 3.8 – *Vue d'ensemble de l'architecture distribuée du modèle à interactions distribuées*

Notre modèle à interactions distribuées se place dans un environnement distribué. Ainsi, certains envois de messages sont des envois distants. Or, il est maintenant reconnu que, dans un environnement distribué, une entité centralisant la gestion d'un mécanisme important est un maillon faible et un goulot d'étranglement vis-à-vis de ce mécanisme. Par conséquent, **l'exécution des interactions ne doit pas être centralisée dans un module** (un serveur notamment) mais, au contraire répartie (critère C3.3). Pour ce faire, les interactions doivent intégrer dans leurs comportements celui gérant leur exécution.

3.2.4 Aspects dynamiques

La mise en œuvre des interactions sous la forme d'objets permet de dissocier l'instanciation des interactions de celle des objets interagissants. Ainsi, une interaction peut être instanciée lorsque le comportement d'un ensemble d'objets doit être modifié en fonction de celui défini dans l'interaction. Lors de la destruction de l'interaction, les objets retrouvent leurs comportements initiaux. Nous avons donc une **déclaration dynamique des interactions** (critère C4.1).

CORBA permet la définition dynamique des interfaces des objets distribués grâce à son concept de dépôt d'interfaces. Conic [MKS89] est la seule approche étudiée offrant un réel support dynamique des interactions. Il base, lui aussi, son support dynamique sur le concept de dépôt (cette fois-ci des descriptions formelles des interactions). Par conséquent, nous n'allons pas réinventer la roue et nous allons proposer un **dépôt de schémas d'interactions** disponible à l'exécution et à distance.

Ce dépôt de schémas d'interactions permettra notamment la définition lors de l'exécution de nouveaux schémas d'interactions (qui pourront ensuite être instanciés en tant qu'interactions) et la modification ou la suppression des schémas d'interactions déjà dans le dépôt de schémas d'interactions (vérification des critères C4.2 et C4.3).

Il permettra également d'offrir à un schéma d'interactions le même niveau conceptuel apporté aux classes par les modèles d'architectures distribuées en fournissant une représentation à l'exécution des schémas d'interactions (de la même manière que l'architecture CORBA fournit une représentation des interfaces IDL). Bien qu'étant représentés par des classes, les schémas d'interactions ne peuvent pas être entièrement décrits par les méta-informations du dépôt d'interface de l'architecture distribuée sous-jacente. En effet, notre modèle à interactions distribuées étend le modèle à objets et les dépôts d'interfaces des architectures distribuées ne sont capables de représenter que les concepts présents dans le modèle à objets.

Partie II

DÉFINITION D'UN MODÈLE À INTERACTIONS ENTRE OBJETS DISTRIBUÉS

Dans cette partie vous trouverez ...

Dans la partie précédente, nous avons étudié les solutions proposées par d'autres travaux pour la prise en compte des interactions et déterminé un ensemble de critères définissant les caractéristiques qui nous semblent essentielles pour un support « idéal » des interactions.

L'objectif de cette partie est la définition d'une part, d'un modèle à interactions distribuées offrant une sémantique claire aux interactions et ce dans un contexte proche des réalités industrielles tel que C++ avec CORBA, mais aussi Java et Java RMI et, d'autre part, d'un langage dédié à la description des interactions : *Interaction Specification Language* (ISL). Nous nous basons sur le modèle à objets défini dans [Jau00] (lui même issu de [San95]) pour modéliser les interactions distribuées.

Avant d'entrer dans le vif du sujet avec la description formelle du modèle, le **chapitre 4** donne une idée concrète de l'utilisation de notre modèle dans le monde compilé et fortement typé qu'est C++ et CORBA. Il explique, en effet, comment définir et enregistrer un schéma d'interactions auprès d'un dépôt de schémas d'interactions (mécanisme permettant une représentation des comportements réactifs à l'exécution) et comment l'instancier sur un ensemble d'objets, puis présente ces différentes étapes de la vie d'un schéma d'interactions à travers un exemple de gestion des feux (tricolores et pour piétons) d'un croisement.

Avec le **chapitre 5**, nous entrons dans le vif du sujet. Ce chapitre fournit une description formelle de notre modèle, que nous appelons modèle à interactions distribuées, en se basant sur un modèle à objets distribués. Il montre l'intégration et la réutilisation qui est faite des principaux concepts du modèle sous-jacent. Il décrit également la sémantique des comportements réactifs.

Ensuite, le **chapitre 6** décrit la fusion comportementale des règles d'interaction. Il présente les règles de réécriture (en sémantique naturelle) qui décrivent la sémantique de la fusion comportementale des règles d'interaction (cette sémantique est utilisée par le mécanisme d'héritage des interactions et lors de l'exécution des comportements réactifs) et démontre les deux propriétés fondamentales de cette fusion, à savoir sa commutativité et son associativité vis-à-vis de la sémantique des opérateurs réactifs.

Chapitre 4

Exemples d'utilisation des interactions : cycle de vie

The most noble and profitable invention of all other was that of speech, consisting of names or appelation, and their connection; whereby men register their thoughts, recall them when they are past, and also declare them one to another for mutual utility and conversation; without which there had been amongst men neither Commonwealth, not society, not contract, nor peace, no more than amongst lions, bears and wolves.
— Thomas Hobbes, Leviathan, I, 4

Ce chapitre présente la syntaxe concrète du langage *Interaction Specification Language* (ISL) qui permet de décrire les schémas d'interactions (et donc les interactions) et illustre notre approche à l'aide d'un exemple. Il explique ensuite, à l'aide d'un autre exemple, comment définir et enregistrer un schéma d'interactions auprès du dépôt de schémas d'interactions et comment instancier (poser) un schéma d'interactions sur un ensemble d'objets. Enfin, il présente ces différentes étapes de la vie d'un schéma d'interactions à travers un exemple de gestion des feux (tricolores et pour piétons) d'un croisement.

4.1 Description des interactions : syntaxe concrète du langage ISL

Le tableau 4.1 page suivante présente une syntaxe concrète du langage ISL (la syntaxe abstraite étant décrite dans le chapitre suivant) en utilisant la notation BNF [Nau63, Knu64]. Un exemple de code ISL est présenté ci-dessous, d'autres se trouvent dans les paragraphes suivants (gestion d'un égaliseur graphique et d'un croisement routier).

EXEMPLE. — Cet exemple présente un schéma d'interactions décrivant un distributeur automatique de boissons :

```
 1: interaction DrinkMachine (PushButton push, Container container, DrinkFactory factory, Money money, Display display)
 2: {
 3:   push.Push () -> if money.EnoughMoney () then push.Push () // money.Debit () ; container.RemoveOne ()
 4:                                          else delegate display.NotEnoughMoney () endif,
 5:   money.Insert (int coin) -> money.Insert (coin) ; display.UpdateMoney (coin),
 6:   money.Reset () -> money.Reset () ; display.ResetMoney ()
 7:   container.RemoveOne (int machine), int nItems -> container.RemoveOne (machine) ; if container.UnderLimit ()
 8:                                                    then container.GetFreeCell (out nItems) ; [ container.Fill (nItems) //
 9:                                                      Factory.Serve (nItems) ] endif
10: }
```

Ce schéma d'interactions contient quatre règles réactives. L'avant dernière règle réactive (ligne 6) est déclenchée (c'est-à-dire que son comportement réactif sera exécuté) par l'invocation de la méthode`Reset` sur l'objet interagissant nommé `money`. Elle spécifie que cette méthode sera exécutée (l'invocation`money.Reset ()` dans le comportement réactif, c'est-à-dire après le symbole `->`, spécifie que la méthode `Reset` sera réellement exécutée et non que la règle réactive sera de nouveau déclenchée), puis qu'ensuite (opérateur d'exécution séquentiel, noté`;`) la méthode `ResetMoney` sur l'objet nommé `display` sera elle aussi exécutée. La seconde règle réactive (ligne 5) est similaire.

La première règle réactive (lignes 3 et 4) définit un comportement réactif conditionnel dont la partie `then` est un comportement réactif séquentiel spécifiant que l'élément du conteneur sera enlevé (message `container.RemoveOne`) lorsque le montant de la boisson est débité du monnayeur et que l'action d'appuyer sur le bouton est réalisée.
On peut noter qu'une règle réactive se compose d'un comportement réactif formel (à droite du `->`) et d'un message déclencheur formel (à gauche du `->`). Elle peut aussi définir un ensemble de variables (quatrième règle réactive).

Syntax	→	Interaction { Interaction }
Interaction	→	'interaction' ClassName '(' Member { ',' Member } ')' [Extensions] [Class] '{' Rules '}'
ClassName	→	ident { '::' ident }
Member	→	ClassName ident
Extensions	→	'extend' Extension { ',' Extension }
Extension	→	ClassName '(' ident { ',' ident } ')'
Class	→	'implement' ClassName
Rules	→	Rule { ',' Rule }
Rule	→	LeftSide '->' Reaction
LeftSide	→	InteractingMessage { ',' Variable }
InteractingMessage	→	ident '.' ident '(' [FormalParameter { ',' FormalParameter }] ')'
Type	→	ClassName | 'integer' | 'string' | 'char' | 'float' | 'mailbox'
FormalParameter	→	Type ident
Variable	→	Type ident
Reaction	→	SequenceOperator
SequenceOperator	→	ConcurrencyOperator { ';' SequenceOperator }
ConcurrencyOperator	→	Msg { '//' ConcurrencyOperator }
Msg	→	MsgWaitAssign | ConditionalOperator | DelegateOperator | CatchOperator
	|	ExceptionOperator | '[' Reaction ']'
MsgWaitAssign	→	MsgAssign | Waiting
MsgAssign	→	Message | Assignment
Message	→	Selector '(' [Parameter { ',' Parameter }] ')'
Selector	→	'super' | 'this' '.' ident | ident '.' ident
Parameter	→	ident | 'in' ident | 'out' ident | 'inout' ident
Assignment	→	ident ':=' MsgAssign
Waiting	→	'wait' MsgWaitAssign 'for' ident
ConditionalOperator	→	'if' MsgWait 'then' Reaction ['else' Reaction] 'endif'
MsgWait	→	Message | Waiting
DelegateOperator	→	'delegate' Reaction
CatchOperator	→	'try' Reaction 'catch' '(' ident ')' Reaction { 'catch' '(' ident ')' Reaction }
ExceptionOperator	→	'exception' ident

TAB. 4.1 – *Syntaxe concrète du langage ISL*

4.2 Exemple introductif au modèle à interactions distribuées : égaliseur graphique

Dans ce paragraphe, nous illustrons notre approche avec un exemple d'égaliseur graphique. Celui-ci est composé de plusieurs pistes d'égalisation pouvant être indépendantes les unes des autres, ou, à la demande de l'utilisateur, être liées, toute modification sur une piste étant, dans ce cas, répercutée sur les autres pistes.

Chaque piste d'égalisation est représentée graphiquement sous la forme d'une glissière disposant d'un bouton de réglage. Lors de la conception du programme, les développeurs ont utilisé le modèle Vue-Contrôleur [KP88]. Nous avons donc les objets suivants :

Des contrôleurs : Un objet contrôleur (de classe `Controleur`) définit la structure et le comportement intrinsèque d'une piste d'égalisation. Il définit un niveau sonore, `vol`, et des méthodes permettant l'obtention et la modification de ce niveau sonore : `obtenirVolume`, `fixerVolume`, `modifierVolume`.

Des glissières : Un objet glissière (de classe `Glissiere`) est un objet graphique représentant la plage de valeurs d'un niveau sonore et matérialise le niveau sonore qui lui est associé à l'aide d'une couleur. Il définit une méthode permettant de spécifier le niveau sonore qu'il doit afficher : `afficherVolume`.

Des boutons de réglage : Un objet bouton de réglage (de classe `Bouton`) est un objet graphique représentant un bouton pouvant être déplacé à l'aide d'une souris. Dans cet exemple, un bouton de réglage servira à modifier le niveau sonore d'une piste d'égalisation. Il définit des méthodes permettant de le déplacer et de lui indiquer qu'il est déplacé à l'aide d'une souris : `deplacer`, `enDeplacement`.

Nous allons définir deux schémas d'interactions dont les instances font participer les objets ci-dessus. Pour chaque schéma d'interactions, nous le décrirons en parallèle de manière informelle et à l'aide d'ISL.

Le schéma d'interactions `VisualiserNiveauSonore` spécifie que lorsque le niveau sonore d'un contrôleur est modifié, ou que le bouton est déplacé à l'aide de la souris, il faut aussi réafficher la glissière et déplacer sur celle-ci le bouton de réglage (on supposera que le bouton se situe sur la glissière et donc qu'un réaffichage de la glissière nécessite de réafficher le bouton par dessus) :

```
1:  interaction VisualiserNiveauSonore (Controleur c, Glissiere g, Bouton b)
2:  {
3:    b.enDeplacement (int delta) -> b.enDeplacement (delta) // c.modifierVolume (delta),
4:    c.modifierVolume (int delta) -> c.modifierVolume (delta) ; b.deplacer (c.obtenirVolume ()),
5:    c.modifierVolume (int delta) -> c.modifierVolume (delta) ; g.afficherVolume (c.obtenirVolume ()),
6:    this.init () -> delegate [ g.afficherVolume (c.obtenirVolume ()) // b.deplacer (c.obtenirVolume ()) ]
7:  }
```

La figure 4.1 montre l'exécution de l'interaction `Visu2`, qui est une instance du schéma d'interactions `VisualiserNiveauSonore` entre les objets `c2`, `g2` et `b2`. Lorsqu'un message `enDeplacement` est envoyé au bouton `b2`, un message `modifierVolume` est également envoyé au contrôleur `c2` (comme cela est spécifié par la troisième ligne du schéma d'interactions). L'envoi de ce message `modifierVolume` sur le contrôleur `c2` implique, comme cela est spécifié par les quatrième et cinquième lignes du schéma d'interactions, d'envoyer aussi un message `deplacer` sur le bouton `b2` ainsi qu'un message `afficherVolume` sur la glissière `g2`. La sixième ligne du schéma d'interactions indique quelle est la réaction qui doit être appliquée lors de la création de l'interaction. Notons que les boutons ne participant à aucune interaction reçoivent et traitent les messages normalement. Ainsi l'envoi du message `enDeplacement` sur le bouton `b5` ne déclenche aucun autre envoi de message contrairement au bouton `b2`.

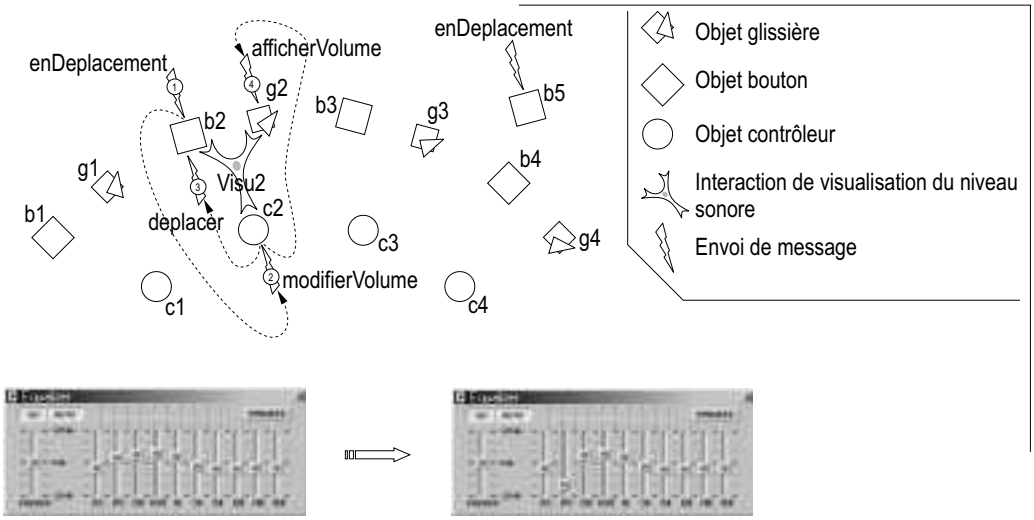


FIG. 4.1 – *Envoi d'un message déclenchant un comportement réactif*

Le schéma d'interactions `LierPistes` spécifie qu'une des pistes peut jouer le rôle de « maître » permettant de modifier le volume général. Lorsque le volume de cette piste est modifié, il doit être répercuté sur les autres pistes. Bien évidemment, la modification du niveau sonore de la piste maître implique de réafficher la glissière et de déplacer sur celle-ci le bouton de réglage. Ainsi ce schéma d'interactions hérite du schéma d'interactions `VisualiserNiveauSonore`. De plus, nous souhaitons que le niveau sonore général puisse être modifié par programmation. Ainsi nous faisons hériter ce schéma d'interactions d'une interface définissant une méthode `modifierVolumeGeneral`. Le schéma est défini comme suit :

```
1:  interaction LierPistes (Controleur c1, Glissiere g, Bouton b,
                            Controleur c2, Controleur c3, Controleur c4)
                            extend VisualiserNiveauSonore (c1, g, b) implement CLierPiste
2:  {
3:    c1.modifierVolume (int delta) -> super (delta) // c2.modifierVolume (delta) //
                                       c3.modifierVolume (delta) // c4.modifierVolume (delta),
4:    this.modifierVolumeGeneral (int delta) -> delegate c1.modifierVolume (delta)
5:  }
```

La règle réactive définie à la ligne 3 du schéma d'interactions `LierPistes` fait appel au mot-clef `super` pour inclure les comportements réactifs associés au même message déclencheur (ici la méthode `modifierVolume` sur l'objet décrit par `c1`) dans le comportement réactif en cours de définition (concept d'héritage des comportements réactifs). Seuls les comportements réactifs peuvent être décrits avec ISL (afin qu'ISL soit indépendant de tout langage de programmation donné). Par conséquent, lorsqu'il est nécessaire de décrire un comportement non réactif, ceci doit être réalisé en faisant appel à un langage à objet. C'est notamment le cas lorsqu'un schéma d'interactions doit intégrer des comportements non réactifs (tel que la méthode `modifierVolumeGeneral`).

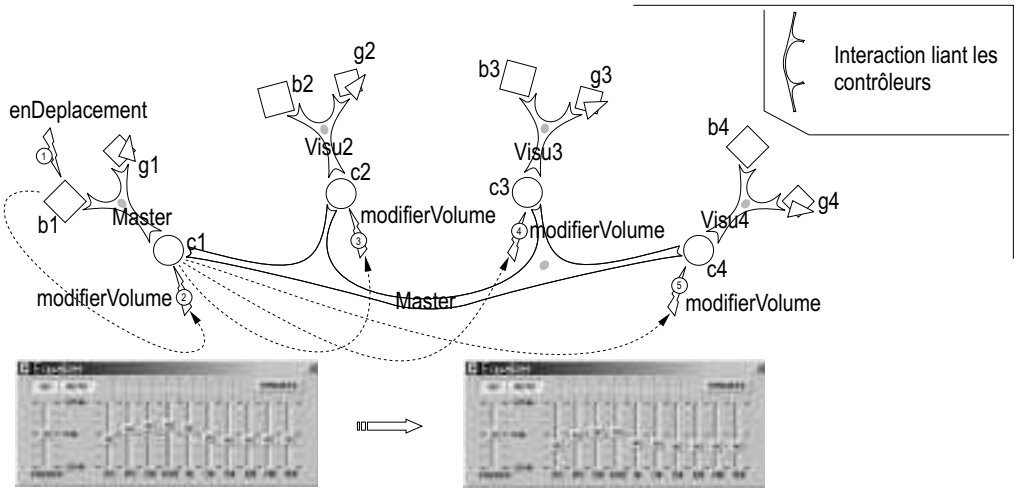


FIG. 4.2 – *Envoi d'un message déclenchant l'exécution de plusieurs interactions en cascade*

La figure 4.2 montre le comportement des objets lorsque les interactions `Visu2`, `Visu3`, `Visu4` ainsi qu'une interaction `Master`, instance du schéma d'interactions `LierPistes`, sont présentes dans le système. Ainsi l'envoi d'un message `enDeplacement` à l'objet `b1` provoque l'envoi d'un message `modifierVolume` à l'objet `c1` grâce à l'interaction `Visu1` (il s'agit de la partie héritée de `LierPistes`). L'envoi de ce message `modifierVolume` sur le contrôleur `c1` implique, comme cela est spécifié par le schéma d'interactions `LierPistes` l'envoi d'un message `modifierVolume` à chacun des autres contrôleurs. Il est à noter que, par souci de clarté, les envois de message impliqués par les règles réactives définies dans les interactions `Visu2`, `Visu3` et `Visu4` ne sont pas représentés sur la figure 4.2.

4.3 De la définition à l'instanciation d'un schéma d'interactions

Nous présentons dans ce paragraphe les différentes étapes pour déclarer une interaction. Nous commencerons par montrer les principes de la définition d'un schéma d'interactions à l'aide du langage ISL. Nous poursuivrons par l'enregistrement du schéma d'interactions dans le dépôt de schémas d'interactions (en décrivant deux solutions différentes). Cet enregistrement d'un schéma d'interactions permet son instanciation, c'est-à-dire la création, entre un ensemble d'objets, interagissants d'une interaction mettant en œuvre les comportements réactifs définis au sein du schéma d'interactions.

4.3.1 Définition d'un schéma d'interactions

La définition d'une interaction se fait en décrivant un schéma d'interactions à l'aide du langage *Interaction Specification Language* (ISL). Cette définition permet de décrire les comportements réactifs des interactions et d'indiquer quels comportements réactifs issus d'autres schémas d'interactions doivent être inclus aux comportements réactifs définis dans le schéma d'interactions en cours de définition (notion d'héritage).

L'utilisation du langage ISL pour décrire les schémas d'interactions (et donc, indirectement, les interactions), permet de dissocier les comportements réactifs définis dans les schémas d'interactions du langage dans lequel sera mise en œuvre l'application qui va instancier le schéma d'interactions. Ainsi cela évite d'avoir à redéfinir les comportements réactifs dans chaque langage à objets pour lesquels des schémas d'interactions vont être instanciés.

Un schéma d'interactions étant mis en œuvre sous la forme d'une classe, le programmeur de ce schéma d'interactions peut définir des comportements (par exemple des méthodes) pour ce schéma

d'interactions. Ceci est réalisé en définissant une classe, dans un langage orienté objets classique, décrivant les comportements non réactifs du schéma d'interactions. Cette classe est associée au schéma d'interactions dans le code ISL du schéma d'interactions grâce au mot-clef `implement`.

L'utilisation d'une classe est une conséquence du fait que, d'une part, les interactions sont des objets (critère C1.1) et, d'autre part, il n'est pas possible de définir des comportements non réactifs en ISL. Cette dernière propriété est un choix que nous avons fait afin de s'assurer que le langage ISL est indépendant de tout langage donné (afin de vérifier le critère C1.3).

4.3.2 Enregistrement du schéma d'interactions dans le dépôt des schémas d'interactions

Une fois le schéma d'interactions défini à l'aide d'ISL, il doit être enregistré auprès du dépôt de schémas d'interactions. Cet enregistrement permet aux applications s'exécutant dans le système de disposer des méta-informations concernant le schéma d'interactions et notamment de celles nécessaires à son instanciation. Comme cela sera détaillé en 8.5.1, le dépôt de schémas d'interactions propose deux types d'accès, l'accès *utilisateur* et l'accès *administrateur*.

Dans le cas d'un accès en tant qu'*utilisateur*, l'application qui enregistre le schéma d'interactions est la seule à pouvoir modifier ou supprimer les méta-informations du schéma d'interactions dans le dépôt de schémas d'interactions, mais toute application du système est en mesure d'instancier le schéma d'interactions. De plus, lorsque l'application cesse d'exister dans le système, tous les schémas d'interactions (et indirectement les interactions instances de ces schémas d'interactions) définis par cette application avec l'accès *utilisateur* sont automatiquement supprimés du dépôt de schémas d'interactions.

Dans le cas d'un accès en tant qu'*administrateur*, l'application qui enregistre le schéma d'interactions partage, aussi bien en lecture qu'en écriture, les méta-informations du schéma d'interactions avec toutes les applications du système. De plus, sauf pour le cas décrit ci-après, les méta-informations du schéma d'interactions ne sont pas automatiquement enlevées du dépôt de schémas d'interactions, ni toutes les interactions instances de ce schéma d'interactions détruites, lorsque l'application cesse d'exister dans le système.

Il est à noter qu'un schéma d'interactions définissant sa propre classe (pour par exemple définir des méthodes et des variables propres à ses instances) et enregistré avec les droits d'accès *administrateur* est automatiquement supprimé du dépôt de schémas d'interactions lorsque l'application qui l'a défini cesse d'exister dans le système car la classe du schéma d'interactions cesse d'exister elle aussi (puisque lors de la suppression de l'application, la description de la classe du schéma d'interactions est automatiquement supprimée du dépôt d'interfaces de l'architecture distribuée sous-jacente).

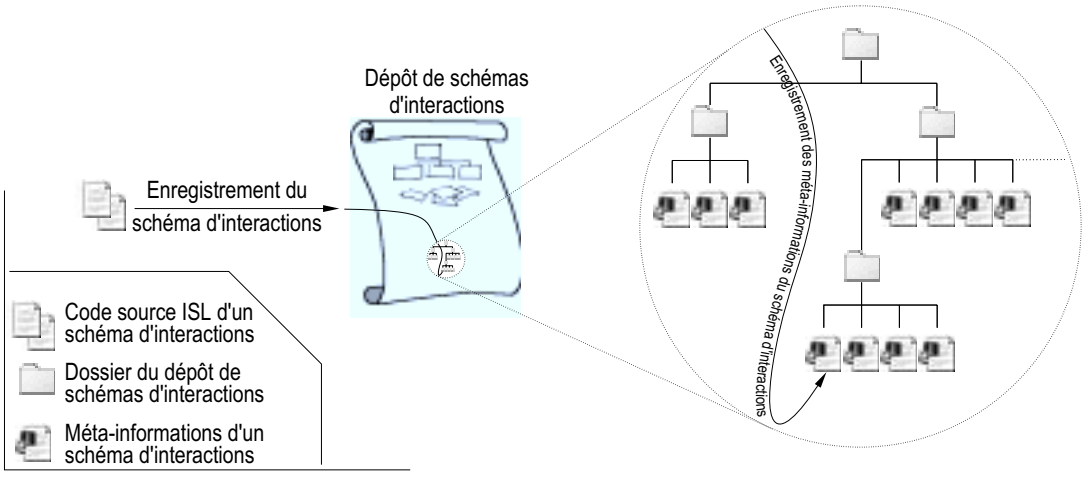


FIG. 4.3 – *Enregistrement d'un schéma d'interactions dans le dépôt de schémas d'interactions*

Dans les deux cas, l'enregistrement d'un schéma d'interactions dans le dépôt de schémas d'interactions implique les trois étapes suivantes (présentées par la figure 4.3) :

- Détermination du dossier dans lequel le schéma d'interactions va être enregistré.
- Création d'une structure de schéma d'interactions permettant de recevoir les méta-informations du schéma d'interactions.
- Inscription dans cette structure des méta-informations décrivant le schéma d'interactions.

4.3.3 Instanciation d'un schéma d'interactions

Une fois le schéma d'interactions enregistré dans le dépôt de schémas d'interactions, n'importe quelle application du système peut l'instancier entre un ensemble d'objets, c'est-à-dire créer une interaction mettant en œuvre les comportements réactifs entre ces objets. Pour ce faire, l'application doit obtenir les méta-informations du schéma d'interactions stocké dans le dépôt de schémas d'interactions.

Ensuite, à partir de ces méta-informations, l'application peut générer une instance du schéma d'interactions en précisant quels objets vont participer à cette instance et donc interagir entre eux. Une fois l'instance du schéma d'interactions créée, l'application peut activer l'interaction, c'est-à-dire modifier effectivement la sémantique de l'envoi de messages pour les messages définis dans les règles réactives de l'interaction afin que les comportements réactifs associés à ces messages soient exécutés en lieu et place du message original.

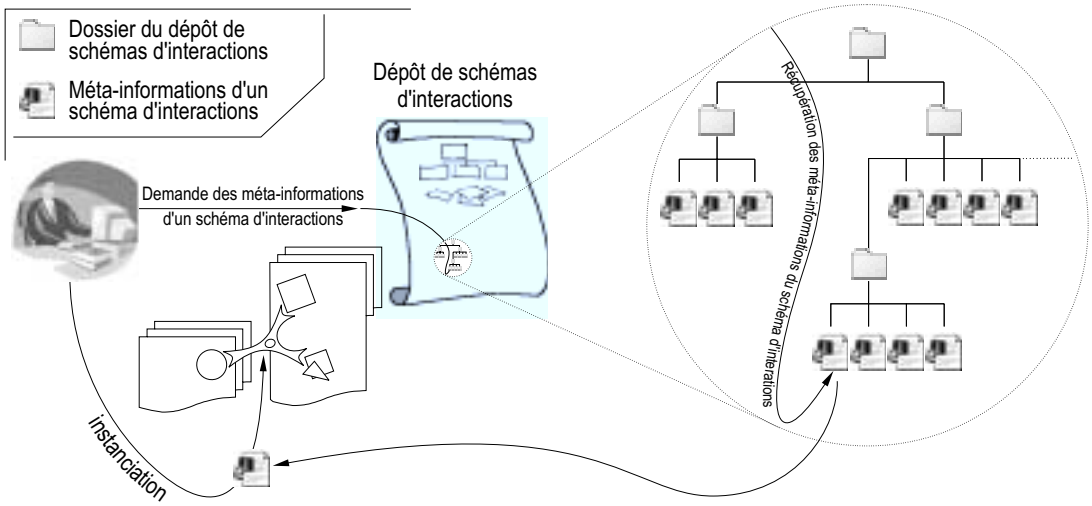


FIG. 4.4 – *Instanciation d'un schéma d'interactions*

Ainsi, pour résumer, les étapes de l'instanciation d'un schéma d'interactions sont les suivantes (présentées par la figure 4.4) :

- Récupérer les méta-informations du schéma d'interactions à instancier à partir du dépôt de schémas d'interactions.
- Déterminer les objets qui vont interagir (c'est-à-dire obtenir leurs identifiants).
- Générer une instance du schéma d'interactions sur les objets ci-dessus. L'interaction est construite dans le dépôt de schémas d'interactions mais ses comportements réactifs sont distribués dans les objets interagissants.
- Activer l'interaction.

4.4 Exemple d'illustration

Ce paragraphe présente, à l'aide d'un exemple de gestion des feux (tricolores et pour piétons) d'un croisement, comment une interaction est définie sous la forme d'un schéma d'interactions puis mise en œuvre entre un ensemble d'objets. Cet exemple sert à définir le vocabulaire et est écrit en ISL et en C++. L'environnement CorbaScript [MGG96] est utilisé pour accéder à l'ORB, le bus logiciel CORBA.

4.4.1 Passage protégé pour piétons

Supposons qu'un programmeur ait défini deux classes indépendantes : la classe `TrafficLight` simulant un feu tricolore et la classe `WalkLight` simulant un feu pour passage pour piétons. Un feu tricolore est un objet possédant au minimum un état fonction de la couleur du feu, une méthode permettant de spécifier la couleur du feu (méthode `setColor`) et trois méthodes permettant de savoir si le feu est vert, orange ou rouge (méthodes `isGreen`, `isYellow` et `isRed`).

Un feu pour piétons est un objet possédant un état indiquant si les piétons peuvent ou non traverser ainsi que deux méthodes permettant de spécifier l'état de l'objet (méthodes `setWalk` et `setDontWalk`). Le programmeur utilise des instances de ces classes normalement. Puis, pour deux de ces instances parti-

culières, `t1` un feu tricolore et `w1` un feu pour piétons, il souhaite exprimer une relation afin de simuler le comportement d'un passage pour piétons protégé par un feu tricolore.

Définition d'un schéma d'interactions

Pour ce faire, le programmeur définit un schéma d'interactions à l'aide du langage ISL (décrit dans le chapitre 5). Il spécifie le comportement associé aux envois de messages sur ces instances à l'aide des opérateurs réactifs : lorsque le feu tricolore change de couleur, si cette nouvelle couleur est le rouge, le feu pour piétons passe à l'état *Walk*, sinon il passe à l'état *Don't Walk*. Il définit aussi un état initial cohérent (afin qu'au moment où l'interaction devient active le feu tricolore et le feu pour piétons se trouvent dans un état valide).

```
1:  interaction TrafficWalk (TrafficLight t, WalkLight w)
2:  {
3:    t.setColor (int color) -> t.setColor (color) ;
                                if t.isRed () then w.setWalk ()
                                              else w.setDontWalk () endif,
4:
5:    this.init () -> delegate if t.isRed () then w.setWalk () else w.setDontWalk () endif
6:  }
```

Sans entrer dans les détails des comportements réactifs, la ligne 3 spécifie que, lorsque le feu tricolore passe au rouge, le feu pour les piétons passe « au vert » (état *Walk*) et que, lorsque le feu tricolore n'est pas rouge, le feu pour piétons est « au rouge » (état *Don't Walk*).

L'initialisation de l'interaction est réalisée par la ligne 5. Elle spécifie que, lorsque la méthode `init` de l'objet représentant l'interaction (objet participant à l'interaction nommé `this`) sera appelée, le feu pour piétons sera dans un état cohérent vis-à-vis de la couleur du feu tricolore.

Enregistrement

Une fois le schéma d'interactions écrit en ISL, le programmeur enregistre ce schéma d'interactions auprès du dépôt de schémas d'interactions, ici en CorbaScript :

```
1:  >>> CORBA.ORB.list_initial_services ()
    ["InterfaceRepository", "NameService", "SchemeRepository", ...]
2:  >>> repository = CORBA.ORB.resolve_initial_references ("SchemeRepository")
3:  >>> name = ["Examples", "CrossRoad"]
4:  >>> repository.bind (name, "interaction TrafficWalk ...")
5:  >>>
```

Tout d'abord il faut obtenir une référence sur le dépôt de schémas d'interactions (ligne 2). Ensuite l'on crée un nom pour le schéma d'interactions que l'on va enregistrer (ligne 3). Finalement l'on enregistre le schéma d'interactions (ligne 4), ici sous la forme d'un source ISL. Une fois cet enregistrement terminé le schéma d'interactions pourra être instancié, modifié, ou supprimé à tout moment.

Pose (ou instanciation)

Une fois le schéma d'interactions enregistré auprès du serveur de schémas d'interactions, le programmeur peut en récupérer une représentation auprès de ce serveur lui permettant de l'instancier :

```
1:  >>> repository = CORBA.ORB.resolve_initial_references ("SchemeRepository")
2:  >>> t1 = CORBA.Object ("IOR:...")
3:  >>> w1 = CORBA.Object ("IOR:...")
4:  >>> name = ["Examples", "CrossRoad", "TrafficWalk"]
5:  >>> scheme = repository.resolve (name)
6:  >>> interaction = scheme.instanciate ( [ t1, w1 ] )
7:  >>>
```

C'est par le biais de cette représentation (les lignes 4 et 5 permettent de la récupérer) que le programmeur peut instancier le schéma d'interactions (ligne 6) en indiquant les instances des objets qui vont participer à l'interaction (obtenus par les lignes 2 et 3). Une fois l'objet interaction créé (ligne 6) les instances `t1` et `w1` sont interagissantes (on appellera de tels objets des objets interagissants).

4.4.2 Feux tricolores : exclusion mutuelle

Continuons avec ce même exemple mais en nous focalisant sur le feu tricolore. Supposons que le programmeur ait défini une classe `Light` représentant une lampe. Il souhaite donc définir le feu tricolore comme étant la composition de trois lampes de couleur rouge, orange et verte. Ainsi, pour les trois instances de lampes qui sont utilisées dans un feu tricolore, le programmeur souhaite exprimer une relation entre elles de telle sorte que, à tout moment, seule une des lampes soit allumée.

Définition d'un second schéma d'interactions

Pour ce faire, il définit un schéma d'interactions à l'aide d'ISL spécifiant une contrainte d'exclusion mutuelle entre les lampes : au plus une seule lampe est allumée à la fois. Il existe une multitude de solutions pour exprimer une telle contrainte. Nous avons choisi ici la plus simple : lorsqu'une lampe reçoit le message qui l'allume on éteint les autres lampes (lignes 6, 7, et 8).

```
1:  interaction TrafficLight (Light green, Light yellow, Light red)
2:    implement CTrafficLight
3:  {
4:    this.setColor (int color) -> this.setColor (color) ;
                                   if this.isGreen () then green.on ()
                                   else if this.isYellow () then yellow.on ()
                                        else red.on () endif endif,
5:
6:    green.on ()  -> [ yellow.off () // red.off ()    ] ; green.on (),
7:    yellow.on () -> [ green.off ()  // red.off ()    ] ; yellow.on (),
8:    red.on ()    -> [ green.off ()  // yellow.off () ] ; red.on ()
9:  }
```

De plus, il définit une classe particulière (décrite ci-après) pour représenter la structure de l'interaction (ligne 2 de la définition ci-dessus du schéma d'interactions). Cette classe déclare les méthodes permettant de connaître l'état du feu et de définir sa couleur (lignes 4, 5, 6, et 7) et hérite de la classe `CScheme` qui définit les comportements par défaut des schémas d'interactions. Ce code est écrit à l'aide d'un langage à objets classique (ici du C++).

```
1:  class CTrafficLight : public CScheme
2:  {
3:    public:
4:      void setColor (int color);
5:      int getColor () const;
6:      bool isGreen () const;
7:      bool isYellow () const;
8:      bool isRed () const;
9:  };
```

L'interface de cette classe sera déposée dans l'*interface repository* de l'architecture distribuée (CORBA dans cet exemple) et sera donc accessible au dépôt de schémas d'interactions lors de l'instanciation du schéma d'interactions.

Enregistrement

Une fois le schéma d'interactions écrit en ISL, le programmeur enregistre ce schéma d'interactions auprès du dépôt de schémas d'interactions :

```
1:  >>> repository = CORBA.ORB.resolve_initial_references ("SchemeRepository")
2:  >>> name = ["Examples", "CrossRoad"]
3:  >>> try {
          repository.bind (name, "interaction TrafficLight ..."); }
        catch (SchemeRepository::AlreadyBound e) {
          println ("Scheme name already bound");
      }
4:  >>>
```

Le programmeur doit aussi enregistrer la classe de l'interaction `CTrafficLight` auprès du système de gestion des objets distribués. CORBA étant l'architecture distribuée utilisée pour cet exemple ceci revient à ajouter la définition de l'interface IDL de la classe `CTrafficLight` dans le dépôt d'interface du bus logiciel de CORBA, l'ORB.

Pose (ou instanciation)

Une fois défini, le schéma d'interactions peut être instancié entre les instances des lampes qui constituent le feu tricolore (description ci-après). Nous vérifions que l'interface de la classe du schéma d'interactions est présente dans le dépôt d'interface de CORBA (ligne 5) en demandant à CorbaScript de nous fournir une description de l'interface (l'absence de l'interface dans le dépôt d'interfaces de CORBA aurait provoqué une exception lors de l'instanciation du schéma d'interactions). Une fois cette vérification effectuée, le schéma d'interactions peut être instancié (ligne 8) sans problème.

Il est à noter que la présence de l'interface effective de l'interaction au sein du dépôt d'interface CORBA est nécessaire à l'instanciation du (ou des) schémas d'interactions basés sur cette interface. Cependant la vérification de la présence de l'interface effective de l'interaction par l'application créant une interaction n'est pas obligatoire puisque l'objet représentant un schéma d'interactions dans le dépôt des schémas d'interactions va en vérifier l'existence et, s'il ne la trouve pas, renvoyer une exception.

```
1:  >>> repository = CORBA.ORB.resolve_initial_references ("SchemeRepository")
2:  >>> green = CORBA.Object ("IOR:...")
3:  >>> yellow = CORBA.Object ("IOR:...")
4:  >>> red = CORBA.Object ("iioploc://dolphin:5000/exemples/...")
5:  >>> SchemesClasses.CTrafficLight
    < OMG-IDL interface SchemesClasses::CTrafficLight {
            booleen isGreen ();
            booleen isYellow ();
            ...
    ); >
6:  >>> name = ["Examples", "CrossRoad", "TrafficLight"]
7:  >>> scheme = repository.resolve (name)
8:  >>> interaction = scheme.instanciate ( [ green, yellow, red ] )
9:  >>>
```

4.4.3 Passage pour piétons protégé par un feu tricolore

Maintenant que nous avons défini le comportement d'un feu tricolore et d'un passage protégé pour piétons nous allons pouvoir les combiner pour définir un passage protégé pour piétons par un feu tricolore. C'est ce que fait le script ci-après.

```
1:  >>> repository = CORBA.ORB.resolve_initial_references ("SchemeRepository")
2:  >>> green = CORBA.Object ("IOR:...")
3:  >>> yellow = CORBA.Object ("IOR:...")
4:  >>> red = CORBA.Object ("iioploc://dolphin:5000/exemples/...")
5:  >>> name = ["Examples", "CrossRoad", "TrafficLight"]
6:  >>> scheme = repository.resolve (name)
7:  >>> trafficlight = scheme.instanciate ( [ green, yellow, red ] )
8:  >>> walker = CORBA.Object ("IOR:...")
9:  >>> name = ["Example", " CrossRoad", "TrafficWalk"]
10: >>> scheme2 = repository.resolve (name)
11: >>> trafficwalk = scheme2.instanciate ( [ trafficlight, walker ] )
12: >>>
```

Ce script se base sur le fait qu'une interaction est un objet et donc qu'il est possible de définir sur celle-ci une interaction (et ainsi définir une interaction dont l'un des participants est lui-même une interaction). Ainsi le script déclare une première interaction, instance du schéma d'interactions `TrafficLight` (présenté dans le paragraphe 4.4.2) et décrivant la sémantique d'un feu tricolore (ligne 6), entre trois instances de lampes (`green`, `yellow` et `red`), et une seconde interaction, instance du schéma d'interactions `TrafficWalk` (présenté dans le paragraphe 4.4.1) décrivant la sémantique entre un feu tricolore et un feu pour piétons. Les objets interagissants pour cette seconde interaction sont, d'une part, la première in-

teraction (décrivant le feu tricolore) et, d'autre part, une instance d'un objet représentant un feu pour piétons (ligne 10).

4.4.4 Gestion du croisement

Nous terminerons la présentation de cet exemple par la définition d'une interaction permettant la gestion complète d'un croisement de deux axes de circulation.

```
1:  interaction SynchronizeLights (TrafficLight t1, TrafficLight t2)
2:  {
3:    t1.setColor (int color) -> t1.setColor (color) // t2.setColor (color),
4:    this.init () -> delegate t2.setColor (t1.getColor ())
5:  }
```

Nous définissons pour ce faire un schéma d'interactions permettant de synchroniser les feux tricolores concernant le même axe de circulation (description ci-dessus). Cette interaction spécifie simplement que lorsque l'état d'un feu tricolore, considéré comme le « feu tricolore maître », change alors l'état de l'autre feu tricolore, considéré comme le « feu tricolore esclave », change lui aussi d'état afin que les deux feux tricolores reflètent le même état. Là aussi, une initialisation de l'interaction est nécessaire afin que les deux feux tricolores possèdent le même état lors de l'activation de l'interaction (ligne 4).

4.5 Autres apports de la réification des interactions

Une interaction, en tant qu'entité de première classe (un objet), peut définir ses propres attributs et comportements (sous la forme de méthodes et variables). De même, il est possible qu'une interaction soit le participant d'une autre interaction.

4.5.1 Comportements propres des interactions : conversion de types

Une interaction étant un objet, elle peut disposer de variables et de comportements propres. Ainsi la réification des interactions offre une solution privilégiée pour représenter les données et les comportements concernant la coordination ou la contrainte entre un ensemble d'objets. Cette référence explicite à l'interaction et à son comportement propre enrichit les comportements réactifs (puisqu'ils peuvent y faire référence) comparée à de simples règles clausales.

Nous verrons, dans le chapitre suivant, que les aspects communs à la vie d'une interaction (création, activation, destruction, etc.) sont gérés par le biais de méthodes définies par la classe de l'interaction. Ceci permet de spécialiser les actions associées aux aspects de la vie d'une interaction (par exemple son activation) en redéfinissant ces méthodes. Cette modification peut être réalisée soit par la définition d'un comportement réactif, soit par la spécialisation de la classe représentant le schéma d'interactions dont l'interaction est une instance.

EXEMPLE. — Dans l'exemple de la gestion d'un croisement routier, le schéma d'interactions`SynchronizeLight` définit un comportement réactif associé à la méthode `init` de l'interaction (mot-clef `this`). Ainsi ce comportement réactif redéfinit le comportement d'initialisation de l'interaction.

En plus de cette adaptation des comportements prédéfinis, il est possible de définir, de manière similaire à la définition des méthodes pour les objets, des comportements spécifiques à une interaction. Cet ajout de comportements à une interaction permet, par exemple, de simuler les concepts du modèle à composants.

Ainsi une interaction peut définir la sémantique de communication entre un ensemble de composants et être considérée elle-même comme un composant. Elle peut aussi servir à adapter les types des paramètres (par conversion de types notamment) entre les messages des objets interagissants.

EXEMPLE. — Supposons que l'on programme un jeu de bataille navale graphique. Il utilise le concept de programmation MVC [KP88]. Ainsi le programmeur définit un objet représentant un échiquier sous la forme d'un tableau unidimensionnel (il s'agit d'un choix de programmation) ainsi qu'un objet graphique représentant cet échiquier.

Il souhaite décrire la cohérence entre l'objet graphique et le tableau grâce à une interaction. Cependant, les méthodes de l'objet graphique utilisent un système de coordonnées à 2 dimensions (x et y) alors que celles du tableau utilisent un système de coordonnées linéaires.

Cependant, cette conversion de types est dépendante du langage applicatif. Elle ne peut donc être décrite à l'aide du langage ISL (puisque ce dernier est indépendant de tout langage de programmation). Heureusement, la conversion de types entre ces deux systèmes de coordonnées peut être associée à l'interaction grâce à la spécialisation (par héritage) de la classe décrivant les schémas d'interactions.

Ainsi, le programmeur peut spécifier, dans la description ISL du schéma d'interactions, la classe qui décrira le schéma d'interactions dans le langage applicatif :

```
interaction NavyBattle (Array array, Graphic graphic) implement ConverteCoord
{
  array.setState (int coord, int state) -> array.setState (coord, state) //
                                           graphic.display (this.getX (coord), this.getY (coord), state)
}
```

Le programmeur peut, de ce fait, définir dans la classe de l'interaction une ou plusieurs méthodes de conversion de types qui seront invoquées dans les règles réactives :

```
class ConvertCoord
{
  int getX (int coord) { ... } const;
  int getY (int coord) { ... } const;
}
```

Il est aussi possible de définir des variables d'instances aux interactions. Ces variables pourront être utilisées par les comportements définis dans l'interaction ou passées comme paramètre des messages dans les règles réactives.

4.5.2 Interactions sur interactions

Les interactions expriment des contraintes de la coordination entre un ensemble d'objets. Or, nous représentons les interactions par des objets. De plus, grâce au langage ISL, il est possible de définir des variables et de comportements propres aux interactions. Ainsi, il est possible d'exprimer des interactions dont certains participants (ou tous les participants) sont des interactions.

Grâce à de telles interactions, il est possible de définir, de manière simple et avec un fort pouvoir d'expression, des propriétés entre des interactions, telles que l'exclusion mutuelle (deux interactions ne sont pas actives en même temps dans le système, etc). Ces interactions peuvent être considérées comme des outils de manipulation des interactions s'intégrant parfaitement dans le modèle à interactions distribuées.

EXEMPLE. — Reprenons l'exemple de l'égaliseur graphique. Nous souhaitons exprimer le fait que les comportements réactifs d'une interaction instance du schéma d'interactions `LierPistes` ne sont actifs dans le système que lorsque l'utilisateur a choisi (grâce à un bouton poussoir) de lier les pistes.

Nous verrons dans le chapitre suivant qu'un schéma d'interactions défini, entre autres, deux comportements (des méthodes), l'un permettant d'activer ses comportements réactifs, l'autre de les désactiver. En notant `activate` et `unactivate` ces comportements, le schéma d'interactions suivant propose une telle sémantique :

```
interaction MutualExclusion (Button button, LierPistes pistes)
{
  button.push () -> button.push () ; if button.isPushed () then pistes.activate () else pistes.unactivate () endif
}
```

4.6 Conclusion

Nous avons montré, dans ce chapitre, comment les interactions étaient utilisées et la manière de décrire les sémantiques des communications à l'aide des schémas d'interactions. Nous avons aussi exposé les relations qui existent entre schémas d'interactions et interactions. Nous allons, dans le chapitre suivant, formaliser tout ceci et décrire de manière détaillée notre modèle à interactions distribuées basé sur le langage ISL.

Chapitre 5

Un modèle à interactions distribuées

Ce chapitre fournit une description formelle de notre modèle, que nous appelons modèle à interactions distribuées, en se basant sur un modèle à objets offrant un support du distribué. Il présente tout d'abord la syntaxe abstraite du langage ISL. Il montre l'intégration et la réutilisation qui est faite des principaux concepts du modèle sous-jacent : héritage, réification et instanciation notamment. Il décrit également la sémantique des comportements réactifs.

NOTE. — Dans ce chapitre, et le suivant, nous décrivons notre modèle indépendamment de tout modèle à objets. En effet, notre modèle à interactions distribuées est basé sur le langage ISL qui est indépendant de tout langage de programmation. Cependant, afin d'employer un vocabulaire connu et de simplifier la définition des propriétés du modèle, nous supposons que le modèle à objets sous-jacent offre la notion de métaclasses et définit les classes comme des objets.

Nous basons nos notations sur celles définies dans [Jau00] (elles mêmes issues de [San95]). Certes ce choix de représentation du modèle à objets sous-jacent est assez éloigné de celui utilisé par les langages que nous visons (C++ et Java en particulier), mais nous verrons dans la partie III que ce choix est malgré tout bien adapté (grâce à l'utilisation de systèmes réflexifs et de la programmation par aspects [KLM+97]).

5.1 Syntaxe abstraite du langage ISL et définitions préliminaires

La syntaxe abstraite décrite dans ce paragraphe utilise le formalisme METAL [KLM83] défini dans CENTAUR [INR89]. Par souci de simplicité la syntaxe abstraite du langage ISL est décomposée en deux sous ensembles, chacun faisant référence à l'autre : un élément en italique dans une syntaxe abstraite indique une référence à un élément de la syntaxe abstraite de l'autre sous-ensemble.

La syntaxe abstraite du tableau 5.1 décrit les éléments de portée globale dans le code ISL, c'est-à-dire les éléments ayant une visibilité de leur point de définition jusqu'à la fin du code ISL. Ces éléments sont des définitions des schémas d'interactions (déclaration `interactingScheme`). Ceux-ci sont constitués d'un ensemble de règles réactives (déclaration `interactingRule`) décrivant des comportements réactifs. La syntaxe abstraite des comportements réactifs est, quant à elle, décrite dans le tableau 5.2. Ceux-ci sont décrits à l'aide d'opérateurs (appelés opérateurs réactifs).

DÉFINITIONS PRÉLIMINAIRES. — Nous notons $\mathcal{A}$ l'ensemble des attributs d'un objet, $\mathcal{O}$ l'ensemble des objets, $\mathcal{N}$ l'ensemble des noms de variables (identificateurs), $\mathcal{C}$ l'ensemble des classes et métaclasses (nous avons $\mathcal{C} \subset \mathcal{O}$) et $\mathcal{B}$ l'ensemble des méthodes. Nous notons $P^{fin}(X)$ l'ensemble des parties finies de l'ensemble X et $L^{fin}(X)$ l'ensemble des listes ordonnées et finies de l'ensemble X.

definition of ISL **is**
abstract syntax

schemes	→	`SchemeDef + ...`
intScheme	→	`InteractingScheme`
scheme	→	`SchemeName Objects Inherited InteractingRules`
schemeClass	→	`SchemeName Objects Inherited Class InteractingRules`
interactingObjects	→	`InteractingObject * ...`
typeObject	→	`Path Class Obj`
inheritedSchemes	→	`InheritedScheme * ...`
inheritDeclaration	→	`SchemeName ObjVars`
vars	→	`Obj + ...`
importPath	→	`SchemeName * ...`
rules	→	`InteractingRule + ...`
rule	→	`InteractingMessage Variables` *InteractingBehavior*
message	→	`TypeModifier MessageTarget MessageSelector MessageParameters`
parameters	→	`MessageParameter * ...`
parameter	→	`MessageParam`
modifierParameter	→	`MessageModifier MessageParam`
modeParam	→	`ModeModifier`
modeTypeParam	→	`ModeModifier TypeModifier`
typeParam	→	`TypeModifier`
interactingVariables	→	`InteractingVariable * ...`
typeVar	→	`TypeModifier Variable`
name	→	**implemented as** `Identifier`
object	→	**implemented as** `Identifier`
target	→	**implemented as** `Identifier`
selector	→	**implemented as** `Identifier`
param	→	**implemented as** `Identifier`
mode	→	**atomic** `in inout out`
type	→	**atomic** `void integer char string float mailbox`
var	→	**implemented as** `Identifier`

`ISL`	::=	schemes
`SchemeDef`	::=	intScheme
`InteractingScheme`	::=	scheme schemeClass
`SchemeName`	::=	name
`Objects`	::=	interactingObjects
`InteractingObject`	::=	typeObject
`ObjVars`	::=	vars
`Path`	::=	importPath
`Class`	::=	name
`Obj`	::=	object
`Inherited`	::=	inheritedSchemes
`InheritedScheme`	::=	inheritDeclaration
`InteractingRules`	::=	rules
`InteractingRule`	::=	rule
`InteractingMessage`	::=	message
`MessageTarget`	::=	target
`MessageSelector`	::=	selector
`MessageParameters`	::=	parameters
`MessageParameter`	::=	parameter modifierParameter
`MessageParam`	::=	param
`MessageModifier`	::=	modeParam modeTypeParam typeParam
`ModeModifier`	::=	mode
`TypeModifier`	::=	type name
`Variables`	::=	interactingVariables
`InteractingVariable`	::=	var typeVar
`Variable`	::=	var

end definition

TAB. 5.1 – *Syntaxe abstraite des schémas d'interactions et des règles réactives*

definition of `InteractingBehavior` **is**
abstract syntax

delegate	→	`InteractingBehavior`
sequential	→	`InteractingBehavior InteractingBehavior`
concurrency	→	`InteractingBehavior InteractingBehavior`
ifThen	→	`StrictMessageCallBehavior InteractingBehavior`
ifThenElse	→	`StrictMessageCallBehavior InteractingBehavior InteractingBehavior`
call	→	`MessageBehavior`
exception	→	`ExceptionId + ...`
try	→	`InteractingBehavior CatchBehaviors`
cascadedCatch	→	`CatchBehavior + ...`
catch	→	`ExceptionId InteractingBehavior`
msg	→	*`MessageTarget MessageSelector`* `MessageArgs`
waiting	→	`MessageBehavior` *`Variable`*
assignment	→	`Variable MessageCallBehavior`
args	→	`MessageArg * ...`
modifierArg	→	*`ModeModifier MessageParam`*
except	→	**implemented as** `String`

`InteractingBehavior`	::=	delegate sequential concurrency ifThen ifThenElse call exception try
`MessageArgs`	::=	args
`MessageArg`	::=	*parameter* modifierArg
`StrictMessageCallBehavior`	::=	msg waiting
`MessageCallBehavior`	::=	msg assignment
`MessageBehavior`	::=	msg waiting assignment
`CatchBehaviors`	::=	cascadedCatch
`CatchBehavior`	::=	catch
`ExceptionId`	::=	except

end definition

TAB. 5.2 – *Syntaxe abstraite des comportements réactifs*

5.2 Règles réactives et comportements réactifs

Dans le chapitre précédent, nous avons introduit les concepts de règles réactives et de comportements réactifs comme moyen de décrire les sémantiques de communications au sein des schémas d'interactions. Dans ce paragraphe et avant de définir formellement les schémas d'interactions, nous définissons la structure des règles réactives ainsi que celle des comportements réactifs. Nous notons $\mathcal{R}$ l'ensemble des règles réactives et $\mathcal{G}$ l'ensemble des comportements réactifs.

5.2.1 Une métaclasse pour les règles réactives

Les règles réactives disposent d'un ensemble d'informations qui leur sont propres, dont notamment le message déclencheur et un comportement réactif. Pour ce faire, le modèle à interactions distribuées définit d'une part les règles réactives sous la forme d'une classe et, d'autre part, une classe spécifique pour toutes les règles réactives (métaclasse). Nous appelons $c_{metarule}$ cette métaclasse.

La classe $c_{metarule}$ des règles réactives définit les trois attributs suivants (il s'agit d'attributs de classe dont la valeur est présente dans l'état des règles réactives) :

- Un attribut $D \in \mathcal{A}$, dont le domaine des valeurs est $\mathcal{C} \times \mathcal{B}$ et dont la valeur est un doublet (P, M), décrivant le message déclencheur de la règle d'interaction instance de la règle réactive. Il spécifie à quel message formel [1] est associé le comportement réactif décrit par la règle réactive.
- Un attribut $CR \in \mathcal{A}$, dont le domaine de valeurs est $\mathcal{G}$, qui sera exécuté en lieu et place du message déclencheur. Ce comportement réactif peut contenir au plus une invocation à la méthode spécifiée par le message déclencheur (afin d'exécuter le comportement initial).
- Un attribut $N \in \mathcal{A}$, dont le domaine de valeurs est $\mathcal{N}$, décrivant, dans le schéma d'interactions où est définie la règle réactive, le participant formel (son nom) déclenchant le comportement réactif CR.

1. Un message formel est un doublet *méthode + classe de l'objet sur lequel s'applique la méthode*. Il se distingue du concept de message du modèle à objets par le fait qu'il n'est pas défini sur un objet particulier.

NOTE. — Il est à noter que les comportements réactifs peuvent être définis sur n'importe quelle méthode publique (méthode d'interface) de n'importe quel objet du système (critères C2.2 et C3.2). Par conséquent, la méthode déclenchante (valeur de *M*) est une méthode publique (d'interface) de la classe (désignée par *P*) de l'objet déclencheur.

EXEMPLE. — Soit la règle réactive (définie dans le schéma d'interactions `VisualiserNiveauSonore` présenté au chapitre 4)
`b.enDeplacement (int delta) -> b.enDeplacement (delta) // c.modifierVolume (delta)`

Le message déclencheur (valeur de l'attribut *D*) est le doublet (`Bouton, enDeplacement (int delta)`), le nom du participant (valeur de l'attribut *N*) est `b` et le comportement réactif (valeur de l'attribut *CR*) est `b.enDeplacement (delta) // c.modifierVolume (delta)`.

5.2.2 Structure d'une règle réactive

La structure de classe des règles réactives définit, entres autres, les trois attributs suivants (attributs d'instance) :

- Un attribut $p \in \mathcal{A}$, dont le domaine de valeurs est $\mathcal{O}$, décrivant l'objet pour lequel l'invocation de la méthode définie par l'attribut de classe *M* va déclencher le comportement réactif.
- Un attribut $cr \in \mathcal{A}$, dont le domaine de valeurs est $\mathcal{G}$, correspondant à l'instanciation de l'attribut de classe *CR* (ce mécanisme d'instanciation est décrit plus en avant dans ce chapitre).
- Un attribut $v \in \mathcal{A}$, dont le domaine de valeurs est $L^{fin}(\mathcal{O})$, correspondant aux valeurs des variables définies dans la règle d'interaction.

De plus, la structure des règles réactives définit le comportement suivant :

- Une méthode d'exécution du comportement réactif *cr*. Cette méthode est appelée lorsque le message déclencheur a été invoqué. Sa sémantique consiste à exécuter le comportement réactif *cr*.

Pour ce faire, le modèle à interactions distribuées définit une classe abstraite [2] c_{rule}, instance de la classe $c_{metarule}$ qui sera héritée par toutes les règles réactives et qui définit ces deux attributs d'instance. Ainsi toute règle réactive hérite de la classe c_{rule} et est une instance de la classe $c_{metarule}$ (ou de l'une de ses sous-classes). La figure 5.1 présente ceci, à l'aide de la notation UML [BJR97], pour le schéma d'interactions `LierPistes` (exemple de l'égaliseur graphique du chapitre précédent).

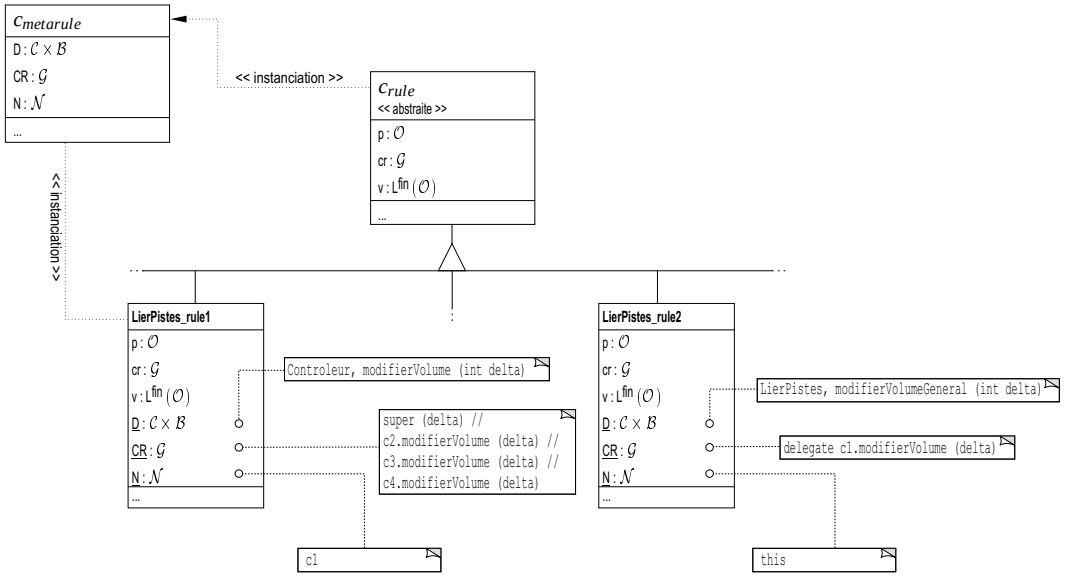


FIG. 5.1 – *Description UML des règles réactives*

5.2.3 Définition des comportements réactifs

Nous notons $\mathcal{G}$ l'ensemble dénombrable des comportements réactifs pouvant être obtenus à partir de la syntaxe abstraite décrite dans le tableau 5.2. Le modèle à interactions distribuées définit, de manière récursive, les comportements réactifs à l'aide de l'ensemble des opérateurs réactifs décrits ci-dessous (chaque opérateur décrit un type de comportement réactif).

2. C'est-à-dire non instanciable.

L'opérateur réactif de délégation (delegate)

Il spécifie qu'un comportement réactif ne comportant pas le message déclencheur de la règle réactive soit traité comme étant ce message déclencheur. Nous notons $\mathcal{G}_D$ l'ensemble des comportements réactifs basés sur l'opérateur réactif de délégation.

L'opérateur réactif séquentiel (sequential)

Il spécifie que deux comportements réactifs doivent être exécutés séquentiellement, c'est-à-dire de manière synchrone. Nous notons $\mathcal{G}_S$ l'ensemble des comportements réactifs basés sur l'opérateur réactif séquentiel.

L'opérateur réactif concurrentiel (concurrency)

Il spécifie que deux comportements réactifs doivent être exécutés concurrentiellement, c'est-à-dire de manière asynchrone (non déterministe). Nous notons $\mathcal{G}_C$ l'ensemble des comportements réactifs basés sur l'opérateur réactif concurrentiel.

L'opérateur réactif d'envoi de messages (msg)

Il spécifie que la méthode qu'il contient doit être exécutée (par envoi de messages). Nous notons $\mathcal{G}_M$ l'ensemble des comportements réactifs basés sur l'opérateur réactif d'envoi de messages.

L'opérateur réactif d'affectation (assignment)

Il spécifie que la valeur d'une variable est affectée à la valeur de retour d'un comportement réactif d'envoi de message ou d'affectation. Nous notons $\mathcal{G}_A$ l'ensemble des comportements réactifs basés sur l'opérateur réactif d'affectation.

L'opérateur réactif d'attente (waiting)

Il spécifie que l'exécution d'un message (par envoi de messages), d'une affectation ou d'une attente est contrainte par la fin de l'exécution d'un message provenant d'un autre comportement réactif (cette fin d'exécution est stockée dans une variable de type ISL `mailbox`). Nous notons $\mathcal{G}_W$ l'ensemble des comportements réactifs basés sur l'opérateur réactif d'attente.

Les opérateurs réactifs conditionnels (ifThen et ifThenElse)

Ils spécifient une exécution conditionnelle d'un comportement réactif en fonction du résultat (un booléen) d'une exécution d'un comportement réactif d'envoi de messages. Nous notons $\mathcal{G}_N$ l'ensemble des comportements réactifs basés sur les opérateurs réactifs conditionnels.

L'opérateur réactif de traitement d'exception (try)

Il spécifie un traitement (sous la forme d'un comportement réactif) à exécuter lorsqu'une exception est détectée dans un comportement réactif. Nous notons $\mathcal{G}_T$ l'ensemble des comportements réactifs basés sur l'opérateur réactif de traitement des exceptions.

L'opérateur réactif d'exception (exception)

Il spécifie un refus d'exécution du message déclencheur. Ce refus pourra être capturé et traité par un *comportement réactif de traitement d'exception*. Nous notons $\mathcal{G}_E$ l'ensemble des comportements réactifs basés sur l'opérateur réactif d'exception et $\mathcal{G}_{\neg E}$ l'ensemble des comportements réactifs basés sur un autre opérateur réactif. Nous avons $\mathcal{G}_{\neg E} = \complement_{\mathcal{G}_E}\mathcal{G}$.

5.2.4 Structure des opérateurs des comportements réactifs

Un opérateur réactif spécifie la sémantique de l'exécution d'un comportement réactif. Nous modélisons donc un comportement réactif par un objet représentant un opérateur réactif. Cette modélisation des comportements réactifs par des objets de première classe permet une spécialisation de la sémantique de ces comportements et l'ajout de nouveaux comportements réactifs (sous la forme de nouveaux opérateurs réactifs). Ainsi la structure d'un comportement réactif est la même que celle d'un objet.

Chacun des comportements réactifs dispose de sa propre classe. Celle-ci décrit la sémantique d'exécution de l'opérateur réactif associé et la sémantique de la fusion comportementale (cette sémantique est décrite dans le chapitre suivant sous la forme d'un ensemble de règles de réécriture). Les concepts communs à tous les comportements réactifs sont décrit dans une classe abstraite, nommée $c_{reactive}$, dont les classes des comportements réactifs héritent. La classe $c_{reactive}$ définit les comportements suivants :

– Une méthode d'exécution du comportement réactif. Cette méthode abstraite est appelée lorsque le comportement réactif doit être exécuté. La sémantique exacte de cette méthode est définie plus avant dans ce chapitre, et ce pour chacun des comportements réactifs définis par le modèle à interactions distribuées.

– Une méthode de fusion du comportement réactif avec un autre comportement réactif. Cette fusion comportementale est notamment appliquée lorsqu'un ensemble de comportements réactifs doit être exécuté suite à l'invocation d'un message. La sémantique de cette fusion est décrite au chapitre suivant.

NOTE. — Il est à noter que les attributs définis dans la classe décrivant un comportement réactif sont dépendant de l'opérateur réactif. Par exemple, pour le comportement réactif séquentiel, sa classe définit deux attributs dont les valeurs sont les deux comportements réactifs qui doivent être exécutés séquentiellement.

5.2.5 Règles réactives et comportements réactifs : une métaclasse commune

Une règle réactive ainsi que ses instances, les règles d'interaction, sont des objets définis par le modèle à interactions distribuées pour sa propre utilisation. Il ne s'agit donc pas d'objets applicatifs. De plus, la sémantique de ces objets fait partie intégrante de la sémantique de la gestion des interactions par le modèle. Ainsi, la modification de cette sémantique grâce à des comportements réactifs peut totalement « corrompre » le fonctionnement des interactions.

De ce fait une règle réactive, mais également une règle d'interaction, ne doit pas être en mesure de participer à une interaction. De même, et pour des raisons identiques, un comportement réactif ne doit pas être en mesure de participer à une interaction.

Pour ce faire, le modèle à interactions distribuées définit une métaclasse commune aux règles réactives et aux comportements réactifs qui empêche la déclaration d'interactions dont un objet règle réactive, règle d'interaction ou comportement réactif est l'un des participants. Nous appelons $c_{\,metareactive}$ cette classe.

5.3 Sémantique des comportements réactifs

La sémantique de l'envoi de messages définie par le modèle à objets doit être étendue pour prendre en compte l'exécution des comportements réactifs des règles d'interaction associées à un message. Dans le modèle à objets, cette sémantique consiste à exécuter la méthode spécifiée par le message en lui fournissant l'environnement d'exécution de l'objet sur lequel s'applique la méthode ainsi que la valeur de ses paramètres.

5.3.1 Exécution des comportements réactifs

Dans le modèle à interactions distribuées, la sémantique de l'envoi de messages est conservée lorsqu'aucune règle d'interaction n'est définie sur le message devant être exécuté mais, lorsqu'une ou plusieurs règles d'interaction sont définies sur le message devant être exécuté, la sémantique de ces fonctions d'exécution consiste à exécuter une combinaison des comportement réactifs (nous verrons en 6.4 le mécanisme de cette combinaison) définis par ces règles d'interaction.

Définition 5.1 : Fonction d'exécution
Soit θ la fonction déterminant le comportement réactif associé à un message et dont le domaine de définition est $\mathcal{O} \times \mathcal{B} \to \mathcal{G}$. Soit λ^r la fonction d'exécution d'un comportement réactif et dont le domaine de définition est $\mathcal{G} \times L^{fin}(\mathcal{O}) \to \mathcal{O}$. La fonction d'exécution λ (décrivant la sémantique de l'envoi de messages et définie par le modèle à objets) est redéfinie par le modèle à interactions distribuées, lorsqu'elle est appliquée à un message interagissant, comme suit :

$$\lambda : \left| \begin{array}{rcl} \mathcal{O} \times \mathcal{B} \times L^{fin}(\mathcal{O}) & \to & \mathcal{O} \\ (o,m,a) & \mapsto & \lambda(o,m,a) = \lambda^r(\theta(o,m),a) \end{array} \right. \tag{5.1}$$

□

EXEMPLE. — En reprenant l'exemple de l'égaliseur graphique, lorsque toutes les interactions sont présentes dans le système alors, lorsque le message `c2.modifierVolume` va être invoqué, la fonction θ va être appelée. Elle va tout d'abord déterminer l'ensemble des règles d'interaction qui doivent être exécutées (phase 1) puis fusionner ces règles d'interaction (phase 2).

Le résultat de la phase 1 est l'ensemble composé des deux règles d'interaction suivantes :

```
c2.modifierVolume (int delta) -> c2.modifierVolume (delta) ; b2.deplacer (c2.obtenirVolume ())
c2.modifierVolume (int delta) -> c2.modifierVolume (delta) ; g2.afficherVolume (c2.obtenirVolume ())
```

Le résultat de la phase 2 (et donc de la fonction θ) est le suivant :

```
c2.modifierVolume (int delta) -> c2.modifierVolume (delta) ; b2.deplacer (c2.obtenirVolume ()) //
                                 g2.afficherVolume (c2.obtenirVolume ())
```

5.3.2 Sémantique de l'exécution des comportements réactifs

Les règles d'interaction étant des citoyens de première classe (elles sont des objets, instances des règles réactives), elles possèdent une méthode permettant l'exécution de leurs comportements réactifs. De même, les comportements réactifs sont des objets qui disposent d'une méthode d'exécution. Dans ce paragraphe, nous décrivons la sémantique de cette méthode, que nous nommons λ^r, pour chacun des comportements réactifs.

Lors de l'exécution de son comportement réactif, une règle d'interaction dispose, en plus des paramètres instanciés du message interagissant (qui sont les valeurs des arguments du message déclencheur), d'un ensemble de variables qui lui sont propres. Nous associons donc à l'exécution d'un comportement réactif un environnement d'exécution contenant les valeurs de ces paramètres et variables. Nous notons $\mathcal{E}$ l'ensemble dénombrable des environnements d'exécution. Puisqu'un environnement d'exécution est une liste de paramètres et de variables (tous des objets) nous avons $\mathcal{E} \subset L^{fin}(\mathcal{O})$.

De plus, chaque comportement réactif dispose d'une valeur de retour. Si le message qui a déclenché l'exécution du comportement réactif fait partie de ce comportement réactif alors la valeur de retour de ce dernier correspond à la valeur de retour de l'exécution du message déclencheur. Si le message qui a déclenché l'exécution du comportement réactif n'est pas présent dans le comportement réactif alors ce dernier a comme valeur de retour celle de l'exécution du dernier message du comportement réactif (par dernier message nous entendons le dernier message qui est exécuté).

Les définitions ci-après décrivent, à l'aide de la sémantique naturelle de TYPOL [Des88], et, pour chaque comportement réactif, la méthode d'exécution réactive λ^r qui lui est associé.

Définition 5.2 : Fonction d'exécution réactive (λ^r)
La fonction λ^r est définie comme suit :

$$\lambda^r : \left| \begin{array}{rcl} \mathcal{G} \times L^{fin}(\mathcal{O}) & \rightarrow & \mathcal{O} \cup \{nil\} \\ (cr, a) & \mapsto & \lambda^r(cr, a) \end{array} \right. \tag{5.2}$$

Où a est la liste des paramètres instanciés du message déclencheur.

□

Définition 5.3 : Fonction d'exécution d'un comportement réactif d'envoi de messages
Nous appelons fonction d'exécution d'un comportement réactif d'envoi de messages la fonction associant à un comportement réactif d'envoi de messages (désigné par l'opérateur msg) le résultat de son exécution (ce résultat est renvoyé sous la forme d'un environnement d'exécution).

Sa sémantique est définie par la règle suivante [3] :

$$\frac{env \vdash \text{call}(o, m, \varsigma(o, m, env)) : o', env'}{env \vdash \text{msg}(o, m) : o', env'}$$

où call est la fonction d'envoi de messages du langage cible et ς est une fonction associant à un message (o, m) la valeur de ses paramètres instanciés définis dans l'environnement d'exécution e :

$$\varsigma : \left| \begin{array}{rcl} \mathcal{O} \times \mathcal{M} \times P^{fin}(\mathcal{E}) & \rightarrow & L^{fin}(\mathcal{O}) \\ (o, m, e) & \mapsto & \varsigma(o, m, e) \end{array} \right. \tag{5.3}$$

□

Définition 5.4 : Fonction d'exécution d'un comportement réactif d'affectation
Nous appelons fonction d'exécution d'un comportement réactif d'affectation la fonction associant à un comportement réactif d'affectation (désigné par l'opérateur assignment) le résultat de son exécution (ce résultat est renvoyé sous la forme d'un environnement d'exécution).

Sa sémantique est définie par la règle suivante :

$$\frac{\begin{array}{l} env \vdash cr_1 : o, env' \\ env' \vdash \text{assign}(x, o) : -, - \end{array}}{env \vdash \text{assignment}(x, cr_1) : o, env'}$$

où assign est la fonction d'affectation du langage applicatif.

□

3. Cette règle signifie que l'exécution d'un comportement réactif d'envoi de messages (opérateur réactif msg) dans un environnement d'exécution env implique l'exécution de la fonction call dans ce même environnement. Le résultat de l'exécution du comportement réactif d'envoi de messages est le résultat de la fonction call (l'objet o') et l'environnement d'exécution généré par l'exécution de l'opérateur réactif d'envoi de messages est celui généré par la fonction d'envoi de message du langage cible.

Définition 5.5 : Fonction d'exécution d'un comportement réactif d'attente
Nous appelons fonction d'exécution d'un comportement réactif d'attente la fonction associant à un comportement réactif d'attente (désigné par l'opérateur waiting) le résultat de son exécution (ce résultat est renvoyé sous la forme d'un environnement d'exécution).

Sa sémantique est définie par la règle suivante :

$$\frac{env \vdash \text{wait}\,(mb) : -, env' \quad env' \vdash cr_1 : o, env''}{env \vdash \text{waiting}\,(cr_1, mb) : o, env''}$$

où wait est une fonction qui attend l'exécution d'un message (grâce à la variable « boîte aux lettres » *mb*) et renvoie l'environnement issu de cette exécution (cette fonction ne renvoit aucune valeur).

□

Définition 5.6 : Fonction d'exécution d'un comportement réactif de délégation
Nous appelons fonction d'exécution d'un comportement réactif de délégation la fonction associant à un comportement réactif de délégation (désigné par l'opérateur delegate) le résultat de son exécution (ce résultat est renvoyé sous la forme d'un environnement d'exécution).

Sa sémantique est définie par la règle suivante :

$$\frac{env \vdash cr_1 : o, env'}{env \vdash \text{delegate}\,(cr_1) : o, env'}$$

□

Définition 5.7 : Fonction d'exécution d'un comportement réactif séquentiel
Nous appelons fonction d'exécution d'un comportement réactif séquentiel la fonction associant à un comportement réactif séquentiel (désigné par l'opérateur sequential) le résultat de son exécution (ce résultat est renvoyé sous la forme d'un environnement d'exécution).

Sa sémantique est définie par les trois règles suivantes :

$$\frac{env \vdash cr_1 : o, env' \quad env' \vdash cr_2 : o', env''}{env \vdash \text{sequential}\,(cr_1, cr_2) : o, env'' \mid m \in cr_1} \qquad \frac{env \vdash cr_1 : o, env' \quad env' \vdash cr_2 : o', env''}{env \vdash \text{sequential}\,(cr_1, cr_2) : o', env'' \mid m \in cr_2}$$

$$\frac{env \vdash cr_1 : o, env' \quad env' \vdash cr_2 : o', env''}{env \vdash \text{sequential}\,(cr_1, cr_2) : o', env'' \mid m \notin cr_1 \wedge m \notin cr_2}$$

□

Définition 5.8 : Fonction d'exécution d'un comportement réactif concurrentiel
Nous appelons fonction d'exécution d'un comportement réactif concurrentiel la fonction associant à un comportement réactif concurrentiel (désigné par l'opérateur concurrency) le résultat de son exécution (ce résultat est renvoyé sous la forme d'un environnement d'exécution).

Sa sémantique est définie par les trois règles suivantes :

$$\frac{env \vdash cr_1 : o, env' \quad env \vdash cr_2 : o', env''}{env \vdash \text{concurrency}\,(cr_1, cr_2) : o, env' \cup env'' \mid m \in cr_1}$$

$$\frac{env \vdash cr_1 : o, env' \quad env \vdash cr_2 : o', env''}{env \vdash \text{concurrency}\,(cr_1, cr_2) : o', env' \cup env'' \mid m \in cr_2}$$

$$\frac{env \vdash cr_1 : o, env' \quad env \vdash cr_2 : o', env''}{env \vdash \text{concurrency}\,(cr_1, cr_2) : o \vee o', env' \cup env'' \mid m \notin cr_1 \wedge m \notin cr_2}$$

□

Définition 5.9 : Fonction d'exécution d'un comportement réactif conditionnel
Nous appelons fonction d'exécution d'un comportement réactif conditionnel la fonction associant à un comportement réactif conditionnel (désigné par l'opérateur ifThenElse) le résultat de son exécution (ce résultat est renvoyé sous la forme d'un environnement d'exécution).

Sa sémantique est définie par les quatre règles suivantes (où la condition est le message $c = (o_c, m_c)$ et *null* l'absence de valeur de retour) :

$$\frac{env \vdash \text{msg}(o_c, m_c) : vrai, env' \quad env' \vdash cr_1 : o, env''}{env \vdash \text{ifThenElse}(c, cr_1, cr_2) : o, env''} \qquad \frac{env \vdash \text{msg}(o_c, m_c) : vrai, env' \quad env' \vdash cr_1 : o, env''}{env \vdash \text{ifThen}(c, cr_1) : o, env''}$$

$$\frac{env \vdash \text{msg}(o_c, m_c) : faux, env' \quad env' \vdash cr_2 : o, env''}{env \vdash \text{ifThenElse}(c, cr_1, cr_2) : o, env''} \qquad \frac{env \vdash \text{msg}(o_c, m_c) : faux, env'}{env \vdash \text{ifThen}(c, cr_1) : null, env''}$$

□

Définition 5.10 : Fonction d'exécution d'un comportement réactif gérant les exceptions
Le langage ISL permet de capturer des exceptions émises par un comportement réactif ou par un message invoqué depuis un comportement réactif (grâce à l'opérateur réactif msg). Ainsi, le support des exceptions est en grande partie dépendant du langage applicatif. En effet, dans un langage applicatif ne définissant pas le concept d'exception, la mise en place du support des exception du modèle à interactions distribuées ne peut être réalisée.

De ce fait, la sémantique de fonction d'un comportement réactif gérant les exceptions présentée ici n'est valable que pour un langage disposant d'un support des exceptions (C++ ou Java par exemple). Nous supposons que le langage propose une fonction, nommée throw servant à déclencher une exception, ainsi qu'une fonction catch servant à capturer une exception. Nous supposons également que lorsqu'une exception est déclenchée par une méthode, celle-ci retourne l'exception.

Cette sémantique est définie par les trois règles suivantes :

$$\frac{env \vdash \text{throw}(e) : o, env'}{env \vdash \text{exception}(e) : o, env'}$$

$$\frac{env \vdash cr_1 : o, env'}{env \vdash \text{try}(cr_1, e, cr_2) : o, env' \mid e \neq o}$$

$$\frac{env \vdash cr_1 : e, env' \quad env' \vdash cr_2 : o, env''}{env \vdash \text{try}(cr_1, e, cr_2) : o, env''}$$

□

5.4 Définition des schémas d'interactions

Un schéma d'interactions est la description des comportements réactifs d'un ensemble d'interactions. Ces comportements réactifs sont décrits, au sein d'un schéma d'interactions, sous la forme de règles réactives. Celles-ci correspondent aux comportements réactifs pouvant être exécutés lors d'un envoi de messages à l'un des objets interagissants participants à l'interaction.

Afin d'apporter aux interactions le même niveau d'abstraction qu'aux objets, les schémas d'interactions sont formalisés sous la forme de classes (vérification du critère C1.1). Ainsi, nous avons $\mathcal{S} \subset \mathcal{C}$.

5.4.1 Une métaclasse pour les schémas d'interactions

Les schémas d'interactions disposent d'un ensemble d'informations qui leur sont propres : la liste des classes et des noms des participants, l'ensemble des règles réactives (et donc des comportements réactifs), et l'ensemble des schémas d'interactions hérités. Par conséquent, tout comme pour les règles réactives, le modèle à interactions distribuées définit une classe spécifique pour tous les schémas d'interactions (concept de métaclasse puisqu'un schéma d'interactions est une classe). Nous appelons $c_{metascheme}$ cette classe « racine » du graphe d'héritage des métaclasses des schémas d'interactions.

La classe $c_{metascheme}$ des schémas d'interactions définit les deux attributs suivants (il s'agit d'attributs de classe dont la valeur est présente dans l'état des schémas d'interactions) :

- Un attribut $P \in \mathcal{A}$ décrivant la liste ordonnée des classes et des noms des participants. Son domaine de valeurs est $L^{fin}(\mathcal{C} \times \mathcal{N})$.
- Un attribut $R \in \mathcal{A}$ décrivant l'ensemble des règles réactives. Son domaine de valeurs est $P^{fin}(\mathcal{R})$.

EXEMPLE. — Reprenons l'exemple de l'égaliseur graphique (paragraphe 4.2).

La valeur de l'attribut *P* décrivant l'ensemble des classes des objets interagissants participants pour le schéma d'interactions `VisualiserNiveauSonore` est la liste `(( Controleur, c ), ( Glissiere, g ), ( Bouton, b ))`.

Son ensemble de règles réactives (valeur de l'attribut *R*) est l'ensemble composé des quatre règles réactives suivantes :

```
b.enDeplacement (int delta) -> b.enDeplacement (delta) // c.modifierVolume (delta),
c.modifierVolume (int delta) -> c.modifierVolume (delta) ; b.deplacer (c.obtenirVolume ()),
c.modifierVolume (int delta) -> c.modifierVolume (delta) ; g.afficherVolume (c.obtenirVolume ()),
this.init () -> delegate [ g.afficherVolume (c.obtenirVolume ()) // b.deplacer (c.obtenirVolume ()) ]
```

5.4.2 Structure d'un schéma d'interactions

La structure de classe des schémas d'interactions définit, entre autres attributs, les deux attributs suivants (attributs d'instance) :

- Un attribut $p \in \mathcal{A}$, dont le domaine des valeurs est $L^{fin}(\mathcal{O})$, décrivant la liste ordonnée des objets interagissants sur lesquels est instancié le schéma d'interactions.
- Un attribut $r \in \mathcal{A}$, dont le domaine des valeurs est $P^{fin}(\mathcal{O})$, décrivant l'ensemble des règles d'interaction[4].

De plus, la structure des schémas d'interactions définit les quatre comportements suivants :

- Une méthode d'initialisation de l'interaction.
- Une méthode « d'activation » des règles d'interaction de l'interaction dans le système.
- Une méthode d'inhibition des règles d'interaction définies par l'interaction.
- Une méthode de terminaison de l'interaction.

Pour ce faire, le modèle à interactions distribuées définit une classe abstraite c_{scheme}, instance de la classe $c_{metascheme}$ qui sera héritée par tous les schémas d'interactions et qui définit ces deux attributs d'instance. Ainsi tout schéma d'interactions hérite directement ou indirectement de la classe c_{scheme} et est une instance de la classe $c_{metascheme}$ (ou de l'une de ses sous-classes).

Ceci est présenté, à l'aide de la notation UML [BJR97], par la figure 5.2.

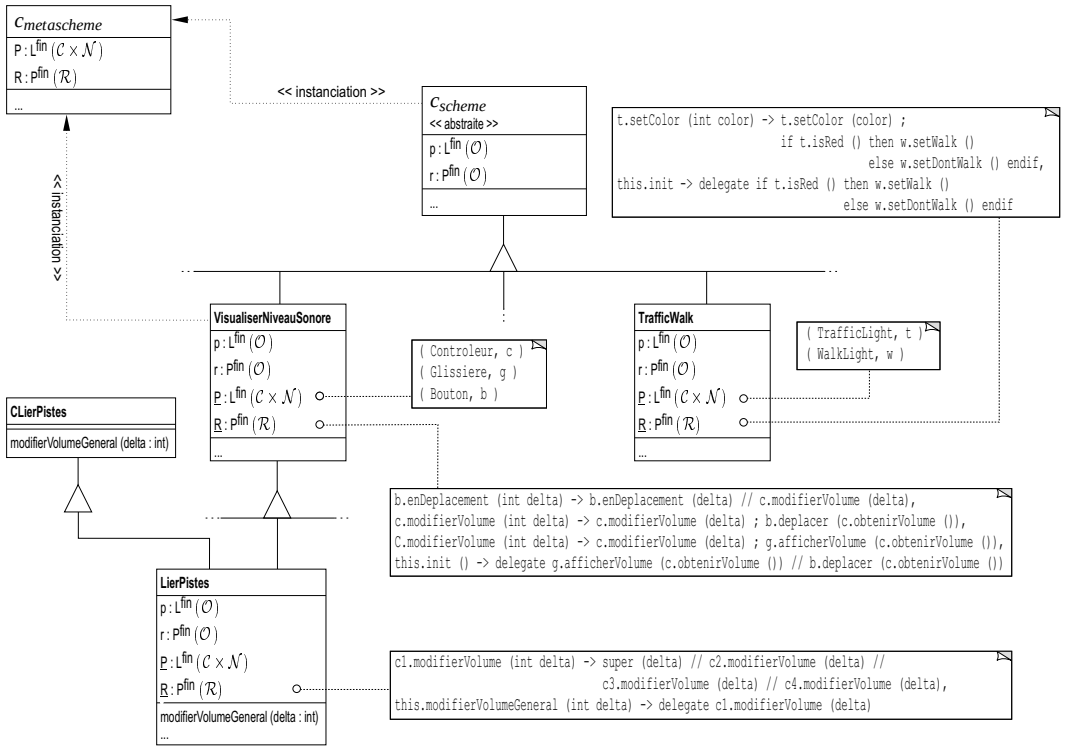


FIG. 5.2 – *Description UML des schémas d'interactions*

4. Nous rappelons qu'une règle d'interaction est une instance d'une règle réactive.

5.4.3 Spécialisation par raffinement

Bien que les schémas d'interactions soient représentés par des classes, l'héritage des règles réactives ne peut être réalisé par le mécanisme d'héritage proposé par le modèle à objets. En effet, les comportements réactifs décrits par les règles réactives sont des attributs définis dans la métaclasse et donc des attributs de classe. De ce fait, leur contenu ne peut donc pas être pris en charge par le concept d'héritage classique du modèle à objets.

Ainsi, la sémantique de cet héritage est définie par le modèle à interactions distribuées et a lieu lors de l'instanciation des schémas d'interactions, afin de valider la cohérence de l'interaction créée (cette sémantique est décrite dans le paragraphe 5.6). Dans ce paragraphe, nous nous restreignons à l'héritage des schémas d'interactions en tant que classes et montrons que la définition d'une classe dérivée d'un schéma d'interactions est un schéma d'interactions.

Définition 5.11 : Fonction d'héritage des schémas d'interactions
Le modèle à objets définit une fonction d'héritage, notée ϖ, associant à une classe (ou une métaclasse) $c \in \mathcal{C}$ l'ensemble des classes dont elle hérite [Jau00] :

$$\varpi : \left| \begin{array}{ccc} \mathcal{C} & \rightarrow & P^{fin}(\mathcal{C}) \\ c & \mapsto & \{c' \in \mathcal{C} \mid c \in \beta(c')\} \end{array} \right. \tag{5.4}$$

Par analogie, le modèle à interactions distribuées définit une fonction d'héritage des schémas d'interactions, notée ϖ_{sch}, qui associe à un schéma d'interactions $s \in \mathcal{S}$ l'ensemble des schémas d'interactions dont il hérite [5] :

$$\varpi_{sch} : \left| \begin{array}{ccc} \mathcal{S} & \rightarrow & P^{fin}(\mathcal{S}) \\ s & \mapsto & \{s' \in S \mid s \in \beta(s')\} = \varpi_{|_{\mathcal{S}}}(s) \end{array} \right. \tag{5.5}$$

Où β est la fonction de sous-classement, définie par le modèle à objets, associant à chaque classe l'ensemble de ses sous-classes et dont le domaine de définition est $\beta : \mathcal{C} \rightarrow P^{fin}(\mathcal{C})$.

□

Soit les deux fonctions suivantes définies par le modèle à objets :

- La fonction d'instanciation, notée γ, qui associe à un objet $o \in \mathcal{O}$ sa classe. Son domaine de définition est $\gamma : \mathcal{O} \rightarrow \mathcal{C}$.
- La fonction de peuplement, notée π^*, qui associe à une classe $c \in \mathcal{C}$ l'ensemble des objets instances de cette classe ou de ses sous-classes. Son domaine de définition est $\pi^* : \mathcal{C} \rightarrow P^{fin}(\mathcal{O})$.

La métaclasse $c_{metascheme}$ étant la « racine » du graphe d'héritage des métaclasses des schémas d'interactions, un schéma d'interactions s vérifie les propriétés suivantes :

- $\forall s \in \mathcal{S}, s \in \pi^*(c_{metascheme})$
- $\forall s \in \mathcal{S}, c_{metascheme} \in \varpi(\gamma(s))$

Ces deux propriétés nous permettent de montrer que les sous-classes d'un schéma d'interactions sont des schémas d'interactions, c'est-à-dire que $\forall s \in \mathcal{S}, \beta(s) \in P^{fin}(\mathcal{S})$.

Propriété 5.1 : Une sous-classe d'un schéma d'interactions est un schéma d'interactions
Toute classe c, sous-classe d'un schéma d'interactions s, est un schéma d'interactions, c'est-à-dire de manière formelle, $\forall s \in \mathcal{S}, \beta(s) \in P^{fin}(\mathcal{S})$.

Preuve :
Soit $s \in \mathcal{S}$ et $\mathcal{E} = \beta(s)$. Soit $c \in \mathcal{E}$. Montrons que $c \in \mathcal{S}$:

- Si $\gamma(c) = c_{metascheme}$ alors $c \in \mathcal{S}$.
- Sinon montrons que $\gamma(c) \in \beta(c_{metascheme})$ et donc, d'après la définition de $c_{metascheme}$, que $c \in \mathcal{S}$.
  Puisque $s \in \mathcal{S}$ nous avons $c_{metascheme} \in \varpi(\gamma(s))$.
  De même puisque $c \in \beta(s)$ nous avons $s \in \varpi(c)$.
  Nous pouvons en déduire (grâce à la propriété de transitivité de ϖ) que $c_{metascheme} \in \varpi(\gamma(c))$ et par conséquent que $c \in \mathcal{S}$.

Nous avons donc $\forall s \in \mathcal{S}, \beta(s) \in P^{fin}(\mathcal{S})$.

□

5. Si E^F dénote l'ensemble des fonctions de E dans F alors $\forall f \in E^F$ et $E' \subseteq E$ alors $f_{|_{E'}}$ dénote la restriction de la fonction f définie sur l'ensemble E'.

5.5 Instanciation des schémas d'interactions

Les schémas d'interactions sont des classes. On peut donc les instancier. Nous appelons ces instances les interactions. Ces dernières sont donc construites à partir des schémas d'interactions en étendant le mécanisme d'instanciation des classes du modèle à objets (puisqu'un schéma d'interactions est une classe). Nous allons donc décrire ce mécanisme et l'adapter aux schémas d'interactions.

5.5.1 Définition formelle de l'instanciation

Définition 5.12 : Fonction d'instanciation des schémas d'interactions
Le modèle à objets définit une fonction d'instanciation, notée γ, qui associe à un objet $o \in \mathcal{O}$ sa classe :

$$\gamma : \left| \begin{array}{ccc} \mathcal{O} & \rightarrow & \mathcal{C} \\ o & \mapsto & \gamma(o) \end{array} \right. \tag{5.6}$$

Par analogie, le modèle à interactions distribuées définit une fonction d'instanciation, notée γ_{sch}, qui associe à une interaction $i \in \mathcal{I}$ son schéma d'interactions :

$$\gamma_{sch} : \left| \begin{array}{ccc} \mathcal{I} & \rightarrow & \mathcal{S} \\ i & \mapsto & \gamma_{sch}(i) = \gamma_{|_{\mathcal{I}}}(i) \end{array} \right. \tag{5.7}$$

□

Définition 5.13 : Fonctions de peuplement des schémas d'interactions
Nous définissons une fonction de peuplement des schémas d'interactions, notée π_{sch}, associant à chaque schéma d'interactions l'ensemble de ses interactions (instances propres) :

$$\pi_{sch} : \left| \begin{array}{ccc} \mathcal{S} & \rightarrow & P^{fin}(\mathcal{I}) \\ s & \mapsto & \pi_{sch}(s) = \{i' \in I \mid s \in \gamma_{sch}(i')\} \end{array} \right. \tag{5.8}$$

De la fonction de peuplement π_{sch} l'on peut définir la fonction d'extension π^*_{sch} associant à chaque schéma d'interactions l'ensemble des interactions créées dans ce schéma d'interactions ou dans ses sous-schémas d'interactions :

$$\pi^*_{sch} : \left| \begin{array}{ccc} \mathcal{S} & \rightarrow & P^{fin}(\mathcal{I}) \\ s & \mapsto & \bigcup\limits_{s' \in \beta(s)} \pi_{sch}(s') \end{array} \right. \tag{5.9}$$

□

Définition 5.14 : Ensemble des interactions
Nous appelons fonction d'ensemble des interactions d'un objet la fonction, notée ι, qui associe à tout objet o l'ensemble des interactions auxquelles il participe :

$$\iota : \left| \begin{array}{ccc} \mathcal{O} & \rightarrow & P^{fin}(\mathcal{I}) \\ o & \mapsto & \iota(o) \end{array} \right. \tag{5.10}$$

□

EXEMPLE. — En reprenant l'exemple de l'égaliseur graphique (paragraphe 4.2), lorsque les interactions`Visu2`, `Visu3`, `Visu4` et `Master` sont présentes dans le système alors l'appel de la fonction ι sur l'objet`c2` renvoie l'ensemble (`Visu2`, `Master`). L'appel de la fonction ι sur l'objet `b2` renvoie l'ensemble (`Visu2`).

Instanciation, héritage et typage

Lors de la définition des schémas d'interactions, nous avons indiqué qu'un schéma d'interactions pouvait être spécialisé par raffinement grâce à la notion d'héritage des comportements réactifs des schémas d'interactions. Nous avons également précisé que cet héritage avait lieu lors de l'instanciation d'un schéma d'interactions.

Cet héritage des comportements réactifs implique une vérification de la cohérence des liens d'héritage qui existent entre les schémas d'interactions afin de s'assurer que l'instanciation du schéma d'interactions construira une interaction ayant du sens (notamment au niveau du typage des participants).

Ainsi, le mécanisme d'instanciation doit s'assurer que les types des participants définis dans un schéma d'interactions *s* sont compatibles (au sens du modèle à objets) vis-à-vis des types des participants définis dans les super-schémas d'interactions de *s*. Cette vérification est dépendante du langage applicatif. Notre modèle à interactions distribuées délègue cette vérification à sa mise en œuvre (se reporter au chapitre 8 pour une description de cette vérification du typage).

5.5.2 Structure des interactions

Les interactions sont des entités capables d'exprimer la communication entre un ensemble d'objets par des comportements réactifs. Ces comportements réactifs sont définis sous forme de règles d'interaction, instances des règles réactives définies dans le schéma d'interactions dont l'interaction est l'instance. Ainsi chaque interaction contient des objets qui représentent ses participants qui peuvent être contactés par envoi de messages à l'aide des règles d'interaction associées à l'interaction.

Toute interaction dispose des deux attributs suivants (définis dans le schéma d'interactions c_{scheme}) :

- Un attribut $p \in \mathcal{A}$, dont le domaine des valeurs est $L^{fin}(\mathcal{O})$, décrivant la liste ordonnée des objets interagissants sur lesquels est instancié le schéma d'interactions (ces objets sont appelés les *participants* de l'interaction). La cardinalité de cette liste est la même que celle de l'attribut de classe *P* décrivant la liste des classes des objets participants du schéma d'interactions. De plus, pour chaque objet participant, son type correspond à celui du même rang défini dans la liste de l'attribut *P*, ou à un type compatible (au sens du langage applicatif).
- Un attribut $r \in \mathcal{A}$, dont le domaine des valeurs est $P^{fin}(\mathcal{O})$, décrivant l'ensemble des règles d'interaction. Cet ensemble correspond à l'instanciation de la combinaison des règles réactives définies dans le schéma d'interactions de l'interaction et de celles définies dans les super-schémas d'interactions de l'interaction. La sémantique de la combinaison est décrite par l'héritage des règles réactives (détaillée dans le paragraphe suivant).

De plus, toute interaction dispose des quatre comportements suivants :

- Une méthode d'initialisation de l'interaction. Cette méthode est appelée juste après la construction de l'objet représentant l'interaction.
- Une méthode « d'activation » des règles d'interaction de l'interaction dans le système.
- Une méthode d'inhibition des règles d'interaction définies par l'interaction. Cette méthode supprime dans le système toutes les règles d'interaction définies par l'interaction sans pour autant détruire l'objet représentant l'interaction.
- Une méthode de terminaison de l'interaction. Cette méthode est appelée juste avant la destruction de l'objet représentant l'interaction.

EXEMPLE. — En reprenant l'exemple de l'égaliseur graphique, l'ensemble des participants pour l'interaction `Visu2` (instance du schéma d'interactions `VisualiserNiveauSonore`) est la liste `( c2, g2, b2 )`. L'ensemble des règles d'interaction de cette interaction est l'ensemble composé de l'instance de chacune des règles réactives définies dans le schéma d'interactions (car ce dernier n'hérite d'aucun schéma d'interactions).
Pour l'interaction `Master`, l'ensemble des participants est la liste `(c1, g1, b1, c2, c3, c4 )`.
L'ensemble des règles d'interaction est le suivant (le paragraphe 5.6 explique comment obtenir cet ensemble) :

```
b.enDeplacement (int delta) -> b.enDeplacement (delta) // c.modifierVolume (delta),
c1.modifierVolume (int delta) -> [ c1.modifierVolume (delta) ; [ b.deplacer (c1.obtenirVolume ()) //
                                    g.afficherVolume (c1.obtenirVolume ()) ] ] // c2.modifierVolume (delta) //
                                  c3.modifierVolume (delta) // c4.modifierVolume (delta),
this.init () -> delegate [ g.afficherVolume (c1.obtenirVolume ()) // b.deplacer (c1.obtenirVolume ()) ]
```

La règle d'interaction dont le message déclencheur est `c1.modifierVolume` est celle définie dans le schéma d'interactions `LierPistes` à laquelle le mot-clef `super` a été remplacé par le comportement réactif résultant de la fusion comportementale (décrite en 6.4) des règles d'interaction définies dans le schéma d'interactions hérité et dont le message déclencheur est aussi `c1.modifierVolume` (au renommage près).

5.5.3 Structure des règles d'interaction

Une règle d'interaction est une instance d'une règle réactive (qui est une classe). Elle est contenue dans une interaction.

Toute règle d'interaction dispose du comportement et des trois attributs suivants :

- Un attribut $p \in \mathcal{A}$, dont le domaine de valeurs est $\mathcal{O}$, décrivant l'objet pour lequel l'invocation de la méthode définie par l'attribut de classe *M* va déclencher le comportement réactif.

– Un attribut $cr \in \mathcal{A}$, dont le domaine de valeurs est $\mathcal{G}$, correspondant à l'instanciation de l'attribut de classe *CR* (ce mécanisme d'instanciation est décrit ci-après).
– Un attribut $v \in \mathcal{A}$, dont le domaine de valeurs est $L^{fin}(\mathcal{O})$, correspondant aux valeurs des variables définies dans la règle d'interaction.
– Une méthode d'exécution du comportement réactif *cr*. Cette méthode est appelée lorsque le message déclencheur a été invoqué. Sa sémantique consiste à exécuter le comportement réactif *cr*.

5.6 Instanciation des règles réactives : héritage des comportements réactifs

Tout comme pour les comportements des objets, les comportements réactifs d'une interaction sont un concept complexe mettant en œuvre à la fois les paradigmes d'instanciation des règles réactives et d'héritage des schémas d'interactions.

Bien que les schémas d'interactions soient représentés par des classes, l'héritage des comportements réactifs ne peut être réalisé par le mécanisme classique d'héritage proposé par le modèle à objets. En effet, les comportements réactifs décrits par les règles réactives sont des attributs statiques dont la sémantique est définie par le modèle à interactions distribuées. De plus, ces règles réactives doivent être instanciées au sein de l'interaction. Ainsi, nous décrivons dans ce paragraphe ce mécanisme d'instanciation des règles réactives permettant l'héritage des comportements réactifs.

L'héritage des comportements réactifs contenus dans un schéma d'interactions *s* se fait en deux étapes :

– Une première étape qui consiste à déterminer l'ensemble des règles réactives qui doivent être instanciées.
– Une seconde étape qui consiste à instancier proprement dit (au sens instanciation du modèle à objets) chacune des règles réactives obtenues à l'étape précédente.

5.6.1 Détermination des règles réactives à instancier

Soit les quatre fonctions suivantes :

– μ_h^r la fonction retournant, pour un schéma d'interactions $s \in \mathcal{S}$, l'ensemble des règles réactives définies dans les schémas d'interactions hérités par *s*. Son domaine de définition est $\mu_h^r : \mathcal{S} \rightarrow P^{fin}(R)$.
– μ_i^r la fonction retournant, pour un schéma d'interactions $s \in \mathcal{S}$, l'ensemble des règles réactives définies dans le schéma d'interactions *s*. Son domaine de définition est $\mu_i^r : \mathcal{S} \rightarrow P^{fin}(R)$.
– υ la fonction retournant, pour un schéma d'interactions $s \in \mathcal{S}$, l'ensemble des règles réactives devant être instanciées. Son domaine de définition est $\upsilon : \mathcal{S} \rightarrow P^{fin}(R)$.
– η la fonction retournant, pour une règle réactive, son message déclencheur. Son domaine de définition est $\eta : \mathcal{R} \rightarrow \mathcal{B}$.

La détermination des règles réactives à instancier est définie par les trois règles suivantes (décrites sous la forme de trois propriétés).

Propriété 5.2 : Comportement réactif non hérité
S'il existe une règle d'interaction *r* définie dans un schéma d'interactions dont le message déclencheur est *m* et s'il n'existe pas de règle d'interaction définie dans un schéma d'interactions hérité par *s* dont le message déclencheur est aussi *m*, la règle d'interaction *r* fait partie de l'ensemble des règles d'interaction contenues dans les interactions instances du schéma d'interactions *s*.

Ceci se traduit formellement comme suit :

Soit $s \in \mathcal{S}$, $a = \mu_h^r(s)$, $b = \mu_i^r(s)$
$\forall r \in b, \nexists r' \in a \mid \eta(r') = \eta(r) \Rightarrow r \in \upsilon(s)$

□

Propriété 5.3 : Héritage d'un comportement réactif non redéfini
S'il existe une règle d'interaction *r* définie dans un schéma d'interactions hérité par un schéma d'interactions *s* dont le message déclencheur est *m* et s'il n'existe pas de règle d'interaction définie dans le schéma d'interactions *s* dont le message déclencheur est aussi *m*, la règle d'interaction *r* fait partie de l'ensemble des règles d'interaction contenues par les interactions instances du schéma d'interactions *s*.

Ceci se traduit formellement comme suit :

Soit $s \in \mathcal{S}$, $a = \mu_h^r(s)$, $b = \mu_i^r(s)$
$\forall r \in a, \nexists r' \in b \mid \eta(r') = \eta(r) \Rightarrow r \in \upsilon(s)$

□

Propriété 5.4 : Héritage d'un comportement réactif redéfini
S'il existe un ensemble de règles d'interaction *e*, dont le message déclencheur est *m*, définies dans les schémas d'interactions hérités par un schéma d'interactions *s* et s'il existe une règle d'interaction définie dans le schéma d'interactions *s* dont le message déclencheur est aussi *m*, une règle d'interaction r', issue de *r*, fait partie de l'ensemble des règles d'interaction contenues par les interactions instances du schéma d'interactions *s*.

En fait cette règle d'interaction r' est identique à la règle d'interaction *r* à laquelle les comportements réactifs définis dans les règles d'interaction de l'ensemble *e* ont été inclus (par application de la fonction de fusion comportementale, décrite en 6.4) à la position, dans *r*, définie par le mot-clef `super` du langage ISL.

Notons $subst(a,b,c)$ la fonction substituant le comportement réactif *c* par le comportement réactif *b* dans le comportement réactif *a* et φ la fonction de fusion comportementale sur les ensembles (cette fonction est décrite au chapitre 6).

L'héritage de comportements réactifs redéfinis se traduit formellement comme suit :

Soit $s \in \mathcal{S}$, $a = \mu_h^r(s)$, $b = \mu_i^r(s)$, $m \in \mathcal{B}$
Soit $e \subseteq a \mid r_1 \in e \Rightarrow \eta(r_1) = m$,
$\forall r \in b, \eta(r) = m \Rightarrow subst(r, \varphi(e), \texttt{super}) \in \upsilon(s)$

□

5.6.2 Algorithme d'héritage des règles réactives

Soient `schéma` un schéma d'interaction et `sur-schémas` l'ensemble des schémas d'interactions dont il hérite. Les règles réactives de `schéma`, après héritage des schémas d'interactions `sur-schémas`, sont spécifiées par la fonction `HériteRéactif` décrite ci-après.

```
HériteRéactif (schéma, sur-schémas)
  H := ∅
  HR := ∅
  Pour tous S de sur-schémas
    HR := Renomme (S, Définition-du-renommage (schéma, S))
  Pour tous R de schéma
    H := Ajoute (H, Fusion-ou-Ajout (R, HR))
  H := Ajoute (HR, H)
```

Les règles réactives issues d'un schéma d'interactions hérité sont renommées (fonction `Renomme`) en fonction des noms des participants dans le schéma d'interactions et le super-schéma d'interactions (fonction `Définition-du-renommage`). Ensuite l'héritage des règles réactives définies dans les super-schémas d'interactions et redéfinies dans le schéma d'interactions est effectuée (fonction `Fusion-ou-Ajout`). Puis les règles réactives héritées non redéfinies dans le schéma d'interactions sont ajoutées à l'ensemble des règles réactives qui seront ensuite instanciées dans l'interaction, instance du schéma d'interactions `schema`.

```
Fusion-ou-Ajout (R, HR)
  HF := ∅
  Pour tous R' de HR
    si Déclenche (R') = Déclenche (R)
       alors HF := Ajoute (R, HF) et HR := Supprime (R', HR)
  RF := Fusion (HF)
  retourne Remplace (R, 'super', RF)
```

Lorsqu'une règle réactive redéfinit des règles réactives héritées alors ces règles réactives redéfinies sont fusionnées entre elles puis le résultat de cette fusion est insérée en lieu et place du mot-clef `super` dans la règle réactive du schéma d'interactions(fonction `Remplace`). Sinon la règle réactive est ajoutée aux règles réactives du schéma d'interactions.

EXEMPLE. — Dans l'exemple de l'égaliseur graphique, le schéma d'interactions `LierPistes` hérite du schéma d'interactions `VisualiserNiveauSonore`. Ainsi, l'ensemble des règles d'interaction de l'interaction `Master` (qui est une instance de `LierPistes`) est l'instanciation de la combinaison des règles réactives définies dans `VisualiserNiveauSonore` avec celles définies dans `LierPistes`.
Par exemple la règle réactive `b.enDeplacement (int delta) -> b.enDeplacement (delta) // c.modifierVolume (delta)` définie par le schéma d'interactions `VisualiserNiveauSonore` n'est pas redéfinie par le schéma d'interactions `LierPistes` et son instanciation fait donc partie de l'ensemble des règles d'interaction des instances de`LierPistes`.

Par contre le schéma d'interactions `VisualiserNiveauSonore` définit deux règles réactives dont le message déclencheur est `c.modifierVolume (int delta)` qui sont redéfinies dans le schéma d'interactions `LierPistes`.

Ainsi, le résultat de la fusion comportementale des deux règles réactives définies dans`VisualiserNiveauSonore` sera incorporé à la règle réactive redéfinie par le schéma d'interactions`LierPistes` en lieu et place du mot-clef `super` (propriété 5.4).

La règle réactive résultante est la suivante :

```
c1.modifierVolume (int delta) -> [ c1.modifierVolume (delta) ; [ b.deplacer (c1.obtenirVolume ()) //
                                     g.afficherVolume (c1.obtenirVolume ()) ] ] // c2.modifierVolume (delta) //
                                  c3.modifierVolume (delta) // c4.modifierVolume (delta),
```

5.7 Conclusion

Dans ce chapitre, nous avons présenté de manière formelle notre modèle à interactions distribuées qui est basé sur l'extension du mécanisme de l'envoi de messages, les concepts de schémas d'interactions, d'interactions et de comportements réactifs.

Nous avons notamment abordé les points suivants :

- L'abstraction offerte aux interactions grâce au concept de schémas d'interactions et sa place par rapport aux classes.
- Les avantages de la réification des interactions, des règles d'interactions et des comportements réactifs sous la forme de citoyens de première classe.
- La place des interactions par rapport aux objets interagissants.
- La définition et la déclaration des interactions et la manière dont les concepts du modèle à objets ont été utilisés et étendus aux schémas d'interactions et aux interactions.
- La sémantique de l'exécution des comportements réactifs et leur définition récursive sous la forme d'opérateurs réactifs.

Le chapitre suivant complète cette définition en décrivant de manière détaillée la sémantique de la fonction de fusion comportementale. Il présente les règles de réécriture (en sémantique naturelle) qui décrivent la sémantique de la fusion comportementale des règles d'interaction (comme nous l'avons vu, cette fusion comportementale est notamment appliquée lors de l'exécution d'un ensemble de règles d'interaction) puis démontre ses deux propriétés fondamentales, à savoir la commutativité et l'associativité.

Chapitre 6

Fusion comportementale des règles d'interaction

> « Coordination *is about the actions for synchronization, communications and service usage and provision.* Computation *is about the actions of processes.* » [CR97]

Ce chapitre présente les règles de réécriture (en sémantique naturelle) qui décrivent la sémantique de la fusion comportementale des règles d'interaction (cette sémantique est utilisée par le mécanisme d'héritage des interactions et lors de l'exécution des comportements réactifs) et démontre les deux propriétés fondamentales de cette fusion, à savoir sa commutativité et son associativité vis-à-vis de la sémantique des opérateurs réactifs. Ces deux propriétés permettent d'appliquer la fonction de fusion comportementale sur la liste des règles d'interaction à fusionner sans avoir à se préoccuper de la manière dont la fonction de fusion comportementale est appliquée (pas de notion d'ordre).

6.1 Motivations de la fusion comportementale des règles d'interaction

Nous présentons dans ce paragraphe le principe de fusion comportementale des règles d'interaction. Le chapitre précédent a déjà apporté quelques motivations et exemples d'utilisation de la fusion comportementale, notamment lors de la définition de l'héritage des comportements réactifs décrits dans les schémas d'interactions. Dans le reste de ce paragraphe, nous utiliserons les définitions de schémas d'interactions suivantes pour illustrer nos propos (issus d'un exemple de distributeur de boissons).

– Schéma d'interactions décrivant un distributeur d'une boisson sans monnayeur ; ce schéma d'interactions matérialise le fait que l'appui sur le bouton poussoir du distributeur fait tomber la boisson du conteneur dans le réceptacle.

```
interaction SimpleDrinkMachine (PushButton button, Container container)
{
  button.Push () -> button.Push () ; container.RemoveOne ()
}
```

– Schéma d'interactions décrivant un distributeur de boissons avec monnayeur ; ce schéma d'interactions matérialise le fait que la boisson ne peut être délivrée que si suffisamment d'argent se trouve dans le monnayeur.

```
interaction DrinkMachine (PushButton push, Container container, Money money)
    extend SimpleDrinkMachine (push, container)
{
  push.Push () -> if money.EnoughMoney () then super () ; money.Debit ()
                                          else delegate money.refund () endif
}
```

– Schéma d'interactions décrivant un distributeur de boissons avec monnayeur et afficheur LCD ; ce schéma d'interactions décrit un distributeur de boissons disposant d'un afficheur LCD permettant d'afficher la somme introduite dans le monnayeur et un ensemble de messages prédéfinis (par exemple en cas de monnaie insuffisante).

```
interaction CompleteDrinkMachine (PushButton push, Container container, Money money, Display display)
    extend DrinkMachine (push, container, money)
{
  push.Push () -> if money.EnoughMoney () then push.Push () // money.Debit ()
                                          else delegate display.NotEnoughMoney () endif,
  money.Insert (int coin) -> money.Insert (coin) ; display.UpdateMoney (coin),
  money.Reset () -> money.Reset () ; display.ResetMoney ()
}
```

– Schéma d'interactions décrivant une connexion entre le distributeur de boissons et l'entreprise qui le gère ; ce schéma d'interactions permet à l'entreprise qui gère un distributeur de boissons d'être automatiquement notifiée lorsque le conteneur du distributeur a atteint un seuil critique (« presque vide » par exemple). De plus, il remplit le conteneur en fonction des stocks disponibles et prévient le service de comptabilité.

```
interaction FactoryManager (Container container, Factory factory, Comptable comptable)
{
  container.RemoveOne (), int nItems ->
    container.RemoveOne () ; if container.UnderLimit () then container.GetFreeCell (out nItems) ;
      [ container.Fill (nItems) // factory.Server (nItems) ] ; comptable.ItemsOut (nItems) endif
}
```

– Schéma d'interactions décrivant le schéma de conception Observateur permettant de faire une « trace » de l'utilisation d'un distributeur de boissons ; ce schéma d'interactions décrit le fait qu'à chaque fois qu'une action spécifique est réalisée sur le distributeur de boissons (ici le fait d'enlever une boisson du conteneur) un observateur est notifié.

```
interaction Observable (Cointainer container, Observer observer)
{
  container.RemoveOne () -> container.RemoveOne () // observer.Update ()
  container.Fill (int nItems) -> container.Fill (nItems) // observer.Update ()
}
```

6.1.1 Fusion comportementale lors de l'héritage des comportements réactifs

Il a été vu dans le paragraphe 5.6 que l'héritage des comportements réactifs impliquait, dans certains cas, une combinaison des comportements réactifs qui, étant définis dans un schéma d'interactions *s*, sont redéfinis dans un schéma d'interactions héritant de *s*.

EXEMPLE. — Le schéma d'interactions `DrinkMachine` redéfinit le comportement réactif dont le message déclencheur est le message formel `push.Push ()`. Cette redéfinition fait appel au mot-clef `super` signifiant qu'il doit être remplacé, lors de l'héritage des règles d'interaction, par le comportement réactif issu de la fusion comportementale des règles d'interaction définies dans les schémas d'interactions hérités (ici, il n'y qu'une seule règle, définie dans `SimpleDrinkMachine`.

Ce mécanisme d'héritage des comportements réactifs est réalisé lors de l'instanciation des schémas d'interactions. Ainsi la fusion comportementale des règles d'interaction est appliquée, dans ce cas, sur des messages formels (c'est-à-dire que les paramètres ne sont pas instanciés).

De plus, le mécanisme de fusion comportementale doit prendre en compte une unification des paramètres et des noms des objets interagissants. En effet, comme cela est le cas entre les schémas d'interactions `SimpleDrinkMachine` et `DrinkMachine`, un schéma d'interactions héritant d'un autre schéma d'interactions peut nommer différemment les objets participants. Cette unification s'apparente à l'unification en logique [SS94].

EXEMPLE. — Le schéma d'interactions `DrinkMachine` utilise le nom `push` pour désigner l'objet interagissant bouton poussoir, alors que le schéma d'interactions `SimpleDrinkMachine`, dont hérite `DrinkMachine`, utilise le nom `button` pour désigner le même objet interagissant. Ainsi, l'une des deux règles d'interaction dont la partie sélecteur du message déclencheur est `Push ()` et dont l'objet interagissant est le bouton poussoir doit être renommée pour unifier les noms utilisés entre les deux règles d'interaction. Au terme de cette unification, les deux règles d'interaction pourront être combinées.

6.1.2 Fusion comportementale lors de l'exécution des comportements réactifs

Un autre moment où la fusion comportementale est nécessaire est lors de l'exécution des comportements réactifs définis sur un message. En effet, supposons que sur un objet *o* soit définies trois règles d'interaction dont le message déclencheur est le même. Lorsque ce message va être invoqué sur cet objet les trois comportements réactifs associés aux trois règles d'interaction doivent être exécutées.

Cependant, si ces trois comportements réactifs impliquent l'exécution du message déclencheur, ce dernier ne doit être exécuté qu'une seule et unique fois, tout en s'assurant que les sémantiques d'exécution décrites dans chacune des règles d'interaction sont respectées. C'est justement le rôle de la fusion comportementale lorsqu'elle est appliquée à ce contexte d'exécution des comportements réactifs.

De plus, en raison de la définition dynamique des interactions, les comportements réactifs associés à un message déclencheur peuvent ne pas être les mêmes lors de deux appels au message déclencheur. Ainsi, la fusion comportementale doit être appliquée, au plus tard, à chaque fois qu'un message déclencheur est invoqué.

REMARQUE. — En fait, l'ensemble des règles d'interaction associées à une méthode pour un objet donné ne peut varier que si l'ensemble des interactions « posées » sur cet objet varie. Il est donc possible de n'appliquer la fusion comportementale entre un ensemble de règles d'interaction définies sur un message donné que si l'ensemble des interactions associées à l'objet qui contient le message a changé.

EXEMPLE. — Supposons qu'une instance i_1 du schéma d'interactions `FactoryManager` soit présente dans le système entre des objets *c* (de type `Container`), *f* (de type `Factory`) et *p* (de type `Comptable`) et qu'une instance du schéma d'interactions `Observable` soit présente entre l'objet *c* et un objet *o* (de type `Observer`). Nous avons donc les règles d'interaction suivantes :

```
c.RemoveOne (), int nItems ->
  container.RemoveOne () ; if c.UnderLimit () then nItems := c.GetFreeCell () ;
    [ c.Fill (nItems) // f.Server (nItems) ] ; p.ItemsOut (nItems) endif
```

```
c.RemoveOne () -> c.RemoveOne () // o.Update ()
```

La première règle d'interaction spécifie que le message déclencheur doit être exécuté puis, une fois l'exécution de ce message terminée (opérateur `sequential`), qu'une réaction conditionnelle doit être exécutée. La seconde règle d'interaction spécifie que le message déclencheur et le message `Update` sur l'objet observateur (`observer`) peuvent être exécutés concurremment.

La fusion comportementale de ces deux règles d'interaction doit conserver ces sémantiques. Ainsi, dans la règle d'interaction obtenue par fusion, l'exécution du comportement réactif conditionnel devra toujours se faire après celle du message déclencheur, l'exécution du message `observer.Update ()` devant toujours être réalisée de manière concurrente à celle du message déclencheur.

Il est à noter qu'aucune contrainte d'exécution n'existe entre les deux règles d'interaction hormis celle définie par le message déclencheur. Par conséquent aucune contrainte d'exécution ne doit être introduite entre le comportement conditionnel et le message `observer.Update ()`. Ainsi ces deux comportements réactifs doivent s'exécuter concurremment.

On peut ainsi déduire que le résultat de la fusion comportementale appliquée à ces deux règles d'interaction est le suivant :

```
c.RemoveOne (), int nItems ->
  [ c.RemoveOne () ; if c.UnderLimit () then nItems := c.GetFreeCell () ;
    [ c.Fill (nItems) // f.Server (nItems) ] ; p.ItemsOut (nItems) endif ] // o.Update ()
```

6.2 Règles d'équivalence

Nous définissons ci-dessous un ensemble de règles d'équivalence des comportements réactifs. Ces règles d'équivalence pourront être appliquées afin de transformer un comportement réactif en un autre comportement réactif sémantiquement équivalent.

REMARQUE. — Ceci signifie que l'exécution du comportement réactif produit par l'une des règles d'équivalence sera sémantiquement identique à celle du comportement réactif initial. Ainsi, des messages exécutés séquentiellement dans la règle d'interaction initiale le sont toujours une fois la transformation appliquée et l'ordre d'application est inchangé. Il en est de même pour les messages exécutés concurrentiellement [1].

Il est à noter que ceci n'implique nullement que les deux comportements réactifs fourniront le même résultat. En effet, l'opérateur `concurrency` introduit du non-déterminisme dans le mécanisme d'exécution des comportements réactifs. Ainsi, si deux méthodes sont exécutées concurrentiellement et qu'elles partagent en écriture une variable, la valeur de cette variable sera indéterminée.

1. Cependant, dans ce cas, l'ordre d'application peut être modifié. Ceci est, sémantiquement parlant, sans incidence.

NOTE. — Certaines de ces règles d'équivalence sont implicitement utilisées par la fonction de fusion comportementale (décrite en 6.4) pour rendre syntaxiquement correct le comportement réactif généré par son application ou pour permettre l'application de l'une des règles de fusion définies.

Transformation T1 :
Transformation d'un comportement réactif séquentiel en un comportement réactif concurrentiel

Cette transformation est appelée par l'une des règles de fusion comportementale (décrite en 6.4). Elle transforme une règle d'interaction décrivant un comportement réactif séquentiel par une règle d'interaction décrivant, grâce à un comportement réactif concurrentiel et un comportement réactif d'attente, la même sémantique comportementale.

- $\forall R \in \mathcal{G}_S : m \rightarrow \text{sequential}\,(cr_1, cr_2),\ (cr_1, cr_2) \in \mathcal{G}^2.$
  $R \equiv m \rightarrow \text{concurrency}\,(\text{assignment}\,(cr_1, x)\,, \text{waiting}\,(cr_2, x))$

Cette transformation génère un comportement réactif syntaxiquement incorrect. L'application des deux transformations suivantes (transformation T2 et T3) permet de le rendre syntaxiquement correct.

Transformation T2 :
Combinaison d'un comportement réactif d'attente avec un autre comportement réactif

Cette transformation est implicitement appelée par la transformation T1 afin de rendre syntaxiquement correct le comportement réactif qu'elle génère.

- $\forall R \in \mathcal{G} : m \rightarrow \text{waiting}\,(\text{delegate}\,(cr)\,, x),\ cr \in \mathcal{G}.$
  $R \equiv m \rightarrow \text{delegate}\,(\text{waiting}\,(cr, x))$
- $\forall R \in \mathcal{G} : m \rightarrow \text{waiting}\,(\text{sequential}\,(cr_1, cr_2)\,, x),\ (cr_1, cr_2) \in \mathcal{G}^2.$
  $R \equiv m \rightarrow \text{sequential}\,(\text{waiting}\,(cr_1, x)\,, cr_2)$
- $\forall R \in \mathcal{G} : m \rightarrow \text{waiting}\,(\text{concurrency}\,(cr_1, cr_2)\,, x),\ (cr_1, cr_2) \in \mathcal{G}^2.$
  $R \equiv m \rightarrow \text{concurrency}\,(\text{waiting}\,(cr_1, x)\,, \text{waiting}\,(cr_2, x))$
- $\forall R \in \mathcal{G} : m \rightarrow \text{waiting}\,(\text{ifThen}\,(c, cr)\,, x),\ c \in \mathcal{G}_M \cup \mathcal{G}_W,\ cr \in \mathcal{G}.$
  $R \equiv m \rightarrow \text{ifThen}\,(\text{waiting}\,(c, x)\,, cr)$
- $\forall R \in \mathcal{G} : m \rightarrow \text{waiting}\,(\text{ifThenElse}\,(c, cr_1, cr_2)\,, x),\ c \in \mathcal{G}_M \cup \mathcal{G}_W,\ (cr_1, cr_2) \in \mathcal{G}^2.$
  $R \equiv m \rightarrow \text{ifThenElse}\,(\text{waiting}\,(c, x)\,, cr_1, cr_2)$
- $\forall R \in \mathcal{G} : m \rightarrow \text{waiting}\,(\text{try}\,(cr_1, e, cr_2)\,, x),\ (cr_1, cr_2) \in \mathcal{G}^2.$
  $R \equiv m \rightarrow \text{try}\,(\text{waiting}\,(cr_1, x)\,, e, cr_2)$

Transformation T3 :
Combinaison d'un comportement réactif d'affectation avec un autre comportement réactif

Cette transformation est implicitement appelée par la transformation T1 afin de rendre syntaxiquement correct le comportement réactif qu'elle génère.

- $\forall R \in \mathcal{G} : m \rightarrow \text{assignment}\,(\text{delegate}\,(cr)\,, x),\ cr \in \mathcal{G}.$
  $R \equiv m \rightarrow \text{delegate}\,(\text{assignment}\,(cr, x))$
- $\forall R \in \mathcal{G} : m \rightarrow \text{assignment}\,(\text{sequential}\,(cr_1, cr_2)\,, x),\ (cr_1, cr_2) \in \mathcal{G}^2.$
  $R \equiv m \rightarrow \text{sequential}\,(cr_1, \text{assignment}\,(cr_2, x))$
- $\forall R \in \mathcal{G} : m \rightarrow \text{assignment}\,(\text{concurrency}\,(cr_1, cr_2)\,, x),\ (cr_1, cr_2) \in \mathcal{G}^2.$
  $R \equiv m \rightarrow \text{concurrency}\,(\text{assignment}\,(cr_1, x')\,, \text{assignment}\,(cr_2, x''))\ \mid x = x' \cup x''$
- $\forall R \in \mathcal{G} : m \rightarrow \text{assignment}\,(\text{ifThen}\,(c, cr)\,, x),\ c \in \mathcal{G}_M \cup \mathcal{G}_W,\ cr \in \mathcal{G}.$
  $R \equiv m \rightarrow \text{ifThen}\,(\text{assignment}\,(c, x)\,, cr)$
- $\forall R \in \mathcal{G} : m \rightarrow \text{assignment}\,(\text{ifThenElse}\,(c, cr_1, cr_2)\,, x),\ c \in \mathcal{G}_M \cup \mathcal{G}_W,\ (cr_1, cr_2) \in \mathcal{G}^2.$
  $R \equiv m \rightarrow \text{ifThenElse}\,(c, \text{assignment}\,(cr_1, x)\,, \text{assignment}\,(cr_2, x))$
- $\forall R \in \mathcal{G} : m \rightarrow \text{assignment}\,(\text{try}\,(cr_1, e, cr_2)\,, x),\ (cr_1, cr_2) \in \mathcal{G}^2.$
  $R \equiv m \rightarrow \text{try}\,(\text{assignment}\,(cr_1, x)\,, e, \text{assignment}\,(cr_2, x))$

Transformation T4 :
Commutativité et associativité des comportements réactifs

- Commutativité de l'opérateur concurrency :
  $\forall R \in \mathcal{G}_C : m \rightarrow \text{concurrency}\,(cr_1, cr_2),\ (cr_1, cr_2) \in \mathcal{G}^2.$
  $R \equiv m \rightarrow \text{concurrency}\,(cr_2, cr_1)$
- Associativité de l'opérateur assignment :
  $\forall R \in \mathcal{G}_A : m \rightarrow \text{assignment}\,(x_1, \text{assignment}\,(x_2, cr)),\ cr \in \mathcal{G}_A \cup \mathcal{G}_M.$
  $R \equiv m \rightarrow \text{assignment}\,(x_2, \text{assignment}\,(x_1, cr))$
- Associativité de l'opérateur waiting :
  $\forall R \in \mathcal{G}_W : m \rightarrow \text{waiting}\,(\text{waiting}\,(cr, mb_1), mb_2),\ cr \in \mathcal{G}_W \cup \mathcal{G}_A \cup \mathcal{G}_M.$
  $R \equiv m \rightarrow \text{waiting}\,(\text{waiting}\,(cr, mb_2), mb_1)$
- Associativité de l'opérateur concurrency :
  $\forall R \in \mathcal{G}_C : m \rightarrow \text{concurrency}\,(\text{concurrency}\,(cr_1, cr_2), cr_3),\ (cr_1, cr_2, cr_3) \in \mathcal{G}^3.$
  $R \equiv m \rightarrow \text{concurrency}\,(cr_1, \text{concurrency}\,(cr_2, cr_3))$
- Associativité de l'opérateur sequential :
  $\forall R \in \mathcal{G}_S : m \rightarrow \text{sequential}\,(\text{sequential}\,(cr_1, cr_2), cr_3),\ (cr_1, cr_2, cr_3) \in \mathcal{G}^3.$
  $R \equiv m \rightarrow \text{sequential}\,(cr_1, \text{sequential}\,(cr_2, cr_3))$
- Associativité de l'opérateur try :
  $\forall R \in \mathcal{G}_S : m \rightarrow \text{try}\,(\text{try}\,(cr_1, e_1, cr_2), e_2, cr_3),\ (cr_1, cr_2, cr_3) \in \mathcal{G}^3.$
  $R \equiv m \rightarrow \text{try}\,(\text{try}\,(cr_1, e_2, cr_3), e_1, cr_2)$

Transformation T5 :
Simplification d'une règle d'interaction contenant un comportement réactif d'exception

L'application de cette transformation permet de simplifier une règle d'interaction contenant un comportement réactif d'exception car ce dernier est « absorbant » vis-à-vis des autres comportements réactifs.

- $\forall R \in \mathcal{G}_S : m \rightarrow \text{sequential}\,(\text{exception}\,(e), cr_2),\ cr_2 \in \mathcal{G}.$
  $R \equiv m \rightarrow \text{exception}\,(e)$
- $\forall R \in \mathcal{G}_D : m \rightarrow \text{delegate}\,(\text{exception}\,(e)).$
  $R \equiv m \rightarrow \text{exception}\,(e)$
- $\forall R \in \mathcal{G}_T : m \rightarrow \text{try}\,(\text{exception}\,(e_1), e_2, cr_1),\ cr_1 \in \mathcal{G},\ e_1 \neq e_2.$
  $R \equiv m \rightarrow \text{exception}\,(e_1)$
- $\forall R \in \mathcal{G}_T : m \rightarrow \text{try}\,(\text{exception}\,(e), e, cr_1),\ cr_1 \in \mathcal{G}.$
  $R \equiv m \rightarrow cr_1$

Transformation T6 :
Distributivité d'un comportement réactif de traitement des exceptions par rapport aux autres comportements réactifs

L'application de cette transformation « distribue » un comportement réactif de traitement des exceptions par rapport à un comportement réactif conditionnel. Nous supposons que l'exécution de la condition (une méthode) ne provoque pas l'exception capturée par l'opérateur try.

- Distributivité de l'opérateur try par rapport à l'opérateur ifThen :
  $\forall R \in \mathcal{G}_T : m \rightarrow \text{try}\,(\text{ifThen}\,(c, cr_1), e, cr_2),\ (cr_1, cr_2) \in \mathcal{G}^2.$
  $R \equiv m \rightarrow \text{ifThen}\,(c, \text{try}\,(cr_1, e, cr_2))$
- Distributivité de l'opérateur try par rapport à l'opérateur ifThenElse :
  $\forall R \in \mathcal{G}_T : m \rightarrow \text{try}\,(\text{ifThenElse}\,(c, cr_1, cr_2), e, cr_3),\ (cr_1, cr_2, cr_3) \in \mathcal{G}^3.$
  $R \equiv m \rightarrow \text{ifThenElse}\,(c, \text{try}\,(cr_1, e, cr_3), \text{try}\,(cr_2, e, cr_3))$

6.3 Substitutions et unifications

Nous appelons **substitution** la fonction substituant, pour un ensemble de couple de noms de variable (o, n), le nom de la variable o par le nom de la variable n. Ainsi, appliquer une substitution S, formée de couples (old_name, new_name), à un comportement réactif revient à substituer les noms des variables old_name du comportement réactif par leurs valeurs associées dans S, à savoir new_name.

Il est à noter que les variables des comportements réactifs sont des variables libres (c'est-à-dire qu'il n'y a pas de lien direct entre elles). Par conséquent, l'application d'une substitution S doit conserver cette propriété.

Nous dirons que deux messages m_1 et m_2 peuvent être **unifiés** si, et seulement si, ils ont la même signature et que leurs arguments peuvent être unifiés deux à deux, c'est-à-dire s'il existe une substitution S telle que $S(m_1) = S(m_2)$ ne produise pas de conflit de conversion des variables.

De même, nous dirons que deux comportements réactifs cr_1 et cr_2 peuvent être unifiés si, et seulement si, il existe une substitution S telle que $S(cr_1) = S(cr_2)$ ne produisant pas de conflit de conversion des variables.

EXEMPLE.— Soit la règle d'interaction suivante :

```
windows.move (int dx, int x) -> menu.move (dx, x) ; scrollbars.move (dx, x)
```

L'application de la substitution $S = \{(dx, x)\}$ à cette règle d'interaction renvoie la règle d'interaction suivante :

```
windows.move (int x, int x) -> menu.move (x, x) ; scrollbars.move (x, x)
```

Nous pouvons constater que, avec cette substitution, les deux variables `dx` et `x` provoquent un conflit de conversion (elles portent désormais le même nom). Cette substitution ne peut donc pas être appliquée à un comportement réactif ou à un message.

Définition 6.1 : Fonction d'unification

Nous appellerons fonction d'unification de deux messages, notée ν_{msg} la fonction suivante ($\mathcal{U}$ est l'ensemble infini dénombrable des substitutions) qui associe à deux messages m_1 et m_2 un message unifié m ainsi qu'une substitution minimale s permettant de passer des messages m_1 et m_2 à m :

$$\nu_{msg} : \left| \begin{array}{rcl} \mathcal{B}^2 & \rightarrow & \mathcal{B} \times \mathcal{U} \\ (m_1, m_2) & \mapsto & \nu_{msg}(m_1, m_2) = (m, s) \\ & & m = s(m_1) \wedge m = s(m_2) \end{array} \right. \tag{6.1}$$

□

Propriété 6.1 : Commutativité de la fonction d'unification

La fonction d'unification ν_{msg} est commutative.

Preuve :

Soit $(m, m') \in \mathcal{B}^2$ et $(s, s') \in \mathcal{U}^2$ tels que $\nu_{msg}(m_1, m_2) = (m, s)$ et $\nu_{msg}(m_2, m_1) = (m', s')$. Nous devons montrer que $m = m'$ et $s = s'$.

Nous avons $m' = s'(m_1) = s'\left(s^{-1}(m)\right)$ et $m = s(m_1) = s\left(s'^{-1}(m')\right)$ donc $\exists s'' \in \mathcal{U} \mid s''(m) = s''(m')$ et, par conséquent, $\exists (m'', s'') \in \mathcal{B} \times \mathcal{U} \mid \nu_{msg}(m, m') = (m'', s'')$.

Nous avons par définition de la fonction ν_{msg} : $m'' = s''(m) = s''(s(m_1))$ et $m'' = s''(m') = s''(s'(m_1))$. Donc $s'' \circ s = s'' \circ s'$ d'où $\underbrace{s''^{-1} \circ s''}_{=Id} \circ s = \underbrace{s''^{-1} \circ s''}_{=Id} \circ s'$, d'où, finalement, $s = s'$, et par conséquent $m = m'$.

□

6.4 Fusion comportementale des règles d'interaction

Cette section décrit la fusion comportementale d'un ensemble de règles d'interaction. Cette fusion génère une unique règle d'interaction dont le comportement réactif est soit inclus en lieu et place du mot-clef `super` lors de l'héritage des règles d'interaction, soit exécutée en lieu et place de l'ensemble des règles d'interaction définies et qui a la même sémantique que cet ensemble de règles d'interaction.

Nous présentons tout d'abord la fonction de fusion comportementale entre deux règles d'interaction. Nous étendons cette définition à une fonction de fusion comportementale n-aire. Nous décrivons ensuite les règles de réécriture, à l'aide de la sémantique naturelle TYPOL [Des88], définissant la sémantique de la fonction de fusion comportementale.

Définition 6.2 : Fusion comportementale

Nous définissons la fonction de fusion comportementale, nommée φ, comme suit :

$$\varphi : \left| \begin{array}{rcl} \mathcal{R}^2 & \rightarrow & \mathcal{R} \\ (r_1, r_2) & \mapsto & r \end{array} \right. \tag{6.2}$$

□

Définition 6.3 : Fusion comportementale n-aire
De la fonction de fusion comportementale φ et de ses propriétés de commutativité et d'associativité (décrites plus en avant dans ce chapitre) nous pouvons en déduire la fonction de fusion comportementale n-aire, notée φ^n, définie récursivement comme suit :

$$\varphi^2 : \left| \begin{array}{rcl} \mathcal{R}^2 & \rightarrow & \mathcal{R} \\ (r_1, r_2) & \mapsto & \varphi(r_1, r_2) \end{array} \right. \tag{6.3}$$

$$\forall n > 2 \qquad \varphi^n : \left| \begin{array}{rcl} \mathcal{R}^n & \rightarrow & \mathcal{R} \\ (r_1, r_2, \ldots, r_n) & \mapsto & \varphi\left(r_1, \varphi^{n-1}(r_2, \ldots, r_n)\right) \end{array} \right. \tag{6.4}$$

□

Les règles de réécriture définissant la sémantique de la fusion comportementale sont décrites ci-dessous. Elles sont présentées par ordre décroissant de priorité. Nous les avons classées en deux catégories : les règles de réécriture terminales (elles renvoient le résultat de la fusion comportementale sans appel récursif et sont donc des axiomes), et les règles de réécriture non terminales (elles sont basées sur un appel récursif). Pour toutes ces règles (hormis la règle R1) nous notons s la substitution permettant d'unifier les comportements réactifs des deux règles d'interaction à fusionner et m le message unifié. De plus, sauf mention contraire, chacun des comportements réactifs à fusionner contient une unique invocation au message déclencheur [2].

6.4.1 Règles de réécriture terminales : axiomes

Règle de fusion R1 : Échec de la fusion des sous-comportements réactifs
Lors de la fusion comportementale de deux comportements réactifs, si la fusion comportementale de leurs sous-comportements réactifs échoue alors la fusion comportementale de ces deux comportements réactifs échoue également. Il y a échec lorsqu'aucune règle de fusion comportementale ne peut être appliquée, ou s'il n'existe aucune substitution permettant d'unifier les deux comportements réactifs à fusionner.

□

Règle de fusion R2 : Fusion des exceptions
SÉMANTIQUE INFORMELLE. — Un comportement réactif d'exception est absorbant vis-à-vis des autres comportements réactifs.

SOLUTION INTUITIVE. — Soit la règle d'interaction suivante décrivant la sémantique d'enregistrement d'un document (objet `doc`) dans une application (objet `appli`) :

```
R1: doc.save () -> if (appli.alreadySaved (doc)) then doc.save () else delegate doc.saveAs () endif
```

L'éditeur de cette application souhaite réaliser une version d'évaluation offrant toutes les fonctionnalités de son application mais ne permettant pas la sauvegarde des documents édités. Il ajoute donc la règle d'interaction suivante à la version d'évaluation :

```
R2: doc.save () -> exception Evaluation
```

Ainsi, pour la version d'évaluation, les deux règles d'interaction ci-dessus sont définies. Par conséquent, la règle d'interaction résultante de la fusion comportementale de ces deux règles doit être :

```
doc.save () -> exception Evaluation
```

Ainsi, la sémantique de la fusion comportementale dont au moins l'un des comportements réactifs est un comportement réactif d'exception est décrite par les deux règles de sémantique naturelle suivantes :

$$(m_1 \rightarrow \text{exception}(e_1), m_2 \rightarrow \text{exception}(e_2)) : m \rightarrow \text{concurrency}(\text{exception}(e_1), \text{exception}(e_2))$$

$$(m_1 \rightarrow cr_1, m_2 \rightarrow \text{exception}(e)) : m \rightarrow \text{exception}(e)$$

□

Règle de fusion R3 : Fusion des délégations
SÉMANTIQUE INFORMELLE. — Un comportement réactif de délégation se substitue au message déclencheur et est considéré comme étant ce dernier.

SOLUTION INTUITIVE. — Cet exemple décrit la fusion comportementale de deux règles d'interaction dont l'une est une délégation. La règle d'interaction désignant la délégation est issue d'un schéma d'interactions décrivant le schéma de conception Adaptateur. Elle spécifie que lorsque le message déclencheur est invoqué sur l'objet adaptateur (nommé `adapter`) alors c'est une méthode de l'objet adapté (nommé `adaptee`) qui est réellement exécutée.

2. Le comportement réactif d'exception ne contient pas d'invocation au message déclencheur. Le comportement réactif conditionnel contient 2 invocations : une pour la branche `then` et une autre pour la branche `else`. L'unique cas ou ceci n'est pas vérifié se produit lorsqu'une des branches (ou les deux) contient un comportement réactif de délégation ou d'exception.

Nous avons donc les deux règles d'interaction suivantes :

```
R1: adapter.Request (int param) -> delegate adaptee.TrueRequest (adapter.Convert (param))
R2: adapter.Request (int param) -> adapter.Request (param) // observer.notify () ; adapter.OtherRequest ()
```

Intuitivement le résultat de la fusion comportementale de ces deux règles d'interaction est la règle d'interaction suivante :

```
adapter.Request (int param) -> delegate adaptee.TrueRequest (adapter.Convert (param)) // observer.notify () ;
                               adapter.OtherRequest ()
```

Ainsi, l'invocation du message déclencheur dans la règle d'interaction `R2` est remplacée par le comportement réactif de délégation de la règle d'interaction `R1`.

Ainsi, la sémantique de la fusion comportementale d'un comportement réactif de délégation avec un autre comportement réactif est décrite par les deux règles de sémantique naturelle suivantes :

$$(m_1 \rightarrow \text{delegate}(cr_1), m_2 \rightarrow cr_2) : m \rightarrow \text{subst}(m, s(\text{delegate}(cr_1)), s(cr_2))$$

$$(m_1 \rightarrow cr_1, m_2 \rightarrow \text{delegate}(cr_2)) : m \rightarrow \text{subst}(m, s(\text{delegate}(cr_2)), s(cr_1))$$

où $\text{subst}(m, cr_1, cr_2)$ est une fonction de substitution dont le domaine de définition est $\text{subst} : \mathcal{B} \times \mathcal{R}^2 \rightarrow \mathcal{R}$. Elle substitue dans le comportement réactif cr_2 le message m par le comportement réactif cr_1.

NOTE. — Cette règle de fusion comportementale est la seule décrivant la fusion comportementale d'un comportement réactif de délégation. Ainsi la fusion comportementale de deux comportements réactifs différents exprimant une délégation sur le même message déclencheur échoue car aucune substitution ne peut être trouvée et aucune règle de fusion comportementale (hormis la règle décrivant un échec de la fusion) ne peut être appliquée.

□

Règle de fusion R4 : Fusion des envois de messages

SÉMANTIQUE INFORMELLE. — Un comportement réactif désignant le message déclencheur est neutre vis-à-vis de la fonction de fusion comportementale.

SOLUTION INTUITIVE. — Soit les deux règles d'interaction suivantes (issues d'un appel récursif à la fonction de fusion comportementale) :

```
R1: square.move (int x, int y) -> square.move (x, y)
R2: square.move (nt dx, int dy) -> square.move (dx, dy)
```

Intuitivement, le résultat de la fusion comportementale appliquée à ces deux règles d'interaction doit être le suivant :

```
square.move (int x, int y) -> square.move (x, y)
```

Ainsi, la sémantique de la fusion comportementale d'un comportement réactif d'envoi de messages avec un autre comportement réactif est décrite par la règle de sémantique naturelle suivante :

$$(m_1 \rightarrow \text{msg}(o, m), m_2 \rightarrow cr) : m \rightarrow s(cr)$$

□

6.4.2 Règles de réécriture non terminales

Ces règles de réécriture de la fusion comportementale réappliquent la fonction de fusion comportementale sur une sous partie des comportements réactifs (principe de récursivité).

Règle de fusion R5 : Fusion d'un comportement réactif d'affectation

SÉMANTIQUE INFORMELLE. — Les comportements réactifs d'affectation s'imbriquent les uns dans les autres et, vis-à-vis des autres comportements réactifs, se comportent comme le comportement réactif d'envoi de messages.

SOLUTION INTUITIVE. — Soit les deux règles d'interaction suivantes (issues d'un appel récursif à la fonction de fusion) :

```
R1: canevas.scale (), int s -> s := canevas.scale ()
R2: canevas.scale (), int f -> f := canevas.scale ()
```

Comme on le ferait avec un langage comme C++, le résultat intuitif de l'application de la fonction de fusion comportementale sur ces deux règles d'interaction est la suivante :

```
R1.2: canevas.scale (), int s, int f -> f := s := canevas.scale ()
```

Supposons maintenant que cette dernière règle d'interaction doive être fusionnée avec la règle d'interaction suivante :

```
R3: canevas.scale () -> try canevas.scale () ; rule.scale () catch ( invalidScale ) Reaction
```

Intuitivement, l'application de la fonction de fusion comportementale sur ces deux règles d'interaction doit remplacer le message déclencheur dans la règle d'interaction `R3` par les affectations de la règle d'interaction `R1.2` :

```
canevas.scale (), int s, int f -> try f := s := canevas.scale () ; rule.scale () catch ( invalidScale ) Reaction
```

Ainsi, un comportement réactif d'affectation se comporte comme un comportement réactif d'envoi de messages.

Ainsi, la sémantique de la fusion comportementale d'un comportement réactif d'affectation avec un autre comportement réactif d'affectation est décrite par la règle de sémantique naturelle suivante :

$$\frac{(m_1 \to cr_1, m_2 \to cr_2) : m \to cr_{1.2}}{(m_1 \to \text{assignment}(x_1, cr_1), m_2 \to \text{assignment}(x_2, cr_2)) : m \to \text{assignment}(x_1, \text{assignment}(x_2, cr_{1.2}))}$$

La sémantique de la fusion comportementale d'un comportement réactif d'affectation avec un comportement réactif séquentiel est décrite par les deux règles de sémantique naturelle suivantes :

$$\frac{(m_1 \to cr_1, m_2 \to \text{assignment}(x, cr_3)) : m \to cr_{1.3}}{(m_1 \to \text{sequential}(cr_1, cr_2), m_2 \to \text{assignment}(x, cr_3)) : m \to \text{sequential}(cr_{1.3}, s(cr_2)) \mid m \in cr_1}$$

$$\frac{(m_1 \to cr_2, m_2 \to \text{assignment}(x, cr_3)) : m \to cr_{2.3}}{(m_1 \to \text{sequential}(cr_1, cr_2), m_2 \to \text{assignment}(x, cr_3)) : m \to \text{sequential}(s(cr_1), cr_{2.3}) \mid m \in cr_2}$$

La sémantique de la fusion comportementale d'un comportement réactif d'affectation avec un comportement réactif de traitement des exceptions est décrite par la règle de sémantique naturelle suivante :

$$\frac{(m_1 \to cr_1, m_2 \to \text{assignment}(x, cr_3)) : m \to cr_{1.3}}{(m_1 \to \text{try}(cr_1, e, cr_2), m_2 \to \text{assignment}(x, cr_3)) : m \to \text{try}(cr_{1.3}, e, s(cr_2))}$$

□

Règle de fusion R6 : Fusion d'un comportement réactif d'attente

SÉMANTIQUE INFORMELLE. — Les comportements réactifs d'attente s'imbriquent les uns dans les autres et encapsulent les comportements réactifs d'affectation. De plus, tout comme les comportements réactifs d'affectation, ils se comportent comme des comportements réactifs d'envoi de messages vis-à-vis des autres comportements réactifs.

SOLUTION INTUITIVE.— Soit les deux règles d'interaction suivantes (issues d'un appel récursif à la fonction de fusion comportementale) :

```
R1: canevas.scale (), mailbox mb1 -> wait canevas.scale () for mb1
R2: canevas.scale (), mailbox mb2 -> wait canevas.scale () for mb2
```

L'application de la fonction de fusion comportementale doit s'assurer que l'exécution du message `canevas.scale ()` ne débute que lorsque les deux attentes sont terminées :

```
canevas.scale (), mailbox mb1, mailbox mb2 -> wait [ wait canevas.scale () for mb2 ] for mb1
```

Ainsi, la sémantique de la fusion comportementale d'un comportement réactif d'attente avec un autre comportement réactif d'attente est décrite par la règle de sémantique naturelle suivante :

$$\frac{(m_1 \to cr_1, m_2 \to cr_2) : m \to cr_{1.2}}{(m_1 \to \text{waiting}(cr_1, mb_1), m_2 \to \text{waiting}(cr_2, mb_2)) : m \to \text{waiting}(\text{waiting}(cr_{1.2}, mb_2), mb_1)}$$

La sémantique de la fusion comportementale d'un comportement réactif d'attente avec un comportement réactif d'affectation est décrite par la règle de sémantique naturelle suivante :

$$\frac{(m_1 \to cr_1, m_2 \to \text{assignment}(x, cr_2)) : m \to cr_{1.2}}{(m_1 \to \text{waiting}(cr_1, mb), m_2 \to \text{assignment}(x, cr_2)) : m \to \text{waiting}(cr_{1.2}, mb)}$$

La sémantique de la fusion comportementale d'un comportement réactif d'attente avec un comportement réactif séquentiel est décrite par les deux règles de sémantique naturelle suivantes :

$$\frac{(m_1 \to cr_1, m_2 \to \text{waiting}(mb, cr_3)) : m \to cr_{1.3}}{(m_1 \to \text{sequential}(cr_1, cr_2), m_2 \to \text{waiting}(mb, cr_3)) : m \to \text{sequential}(cr_{1.3}, s(cr_2)) \mid m \in cr_1}$$

$$\frac{(m_1 \to cr_2, m_2 \to \text{waiting}(mb, cr_3)) : m \to cr_{2.3}}{(m_1 \to \text{sequential}(cr_1, cr_2), m_2 \to \text{waiting}(mb, cr_3)) : m \to \text{sequential}(s(cr_1), cr_{2.3}) \mid m \in cr_2}$$

La sémantique de la fusion comportementale d'un comportement réactif d'attente avec un comportement réactif de traitement des exceptions est décrite par la règle de sémantique naturelle suivante :

$$\frac{(m_1 \to cr_1, m_2 \to \text{waiting}(cr_3, mb)) : m \to cr_{1.3}}{(m_1 \to \text{try}(cr_1, e, cr_2), m_2 \to \text{waiting}(cr_3, mb)) : m \to \text{try}(cr_{1.3}, e, s(cr_2))}$$

□

Règle de fusion R7 : Fusion d'un comportement réactif concurrentiel

SÉMANTIQUE INFORMELLE. — La fusion comportementale favorise l'exécution concurrente de comportements réactifs ne disposant pas, avant application de la fonction de fusion comportementale, de relation d'ordre sur l'exécution de ces comportements réactifs.

SOLUTION INTUITIVE. — Soit les deux règles d'interaction suivantes :

```
R1: square.move (int dx, int dy), int dz -> dz := square.move (dx, dy) // trace.update (dz)
R2: square.move (int x, int y) -> square.move (x, y) // properties.updatePos (x, y)
```

Aucune relation d'ordre n'est définie entre les messages `trace.update` et `properties.updatePos`. Par conséquent, le résultat de la fusion comportementale appliquée à ces deux règles d'interaction ne doit pas en définir une.

Ainsi, après application de la fonction de fusion comportementale, ces deux messages doivent être exécutés concurrentiellement :

```
square.move (int x, int y), int dz -> [ dz := square.move (x, y) // properties.updatePos (x, y) ] //
                                       trace.update (dz)
```

Ainsi, la sémantique de la fusion comportementale d'un comportement réactif concurrentiel avec un autre comportement réactif est décrite par la règle de sémantique naturelle suivante [3] :

$$\frac{(m_1 \rightarrow cr_2, m_2 \rightarrow cr_3) : m \rightarrow cr_{2.3}}{(m_1 \rightarrow \text{concurrency}\,(cr_1, cr_2)\,, m_2 \rightarrow cr_3) : m \rightarrow \text{concurrency}\,(s\,(cr_1)\,, cr_{2.3}) \mid m_1 \in cr_2}$$

□

Règle de fusion R8 : Fusion d'un comportement réactif séquentiel

SÉMANTIQUE INFORMELLE. — La fusion comportementale favorise l'exécution concurrente mais doit conserver les relations d'ordre établies au sein des règles d'interaction par les comportements réactifs séquentiels.

SOLUTION INTUITIVE. — Soit les deux règles d'interaction suivantes :

```
R1: window.display (screen s) -> [ window.display (s) ; title.display (s) ] ; border.display (s)
R2: window.display (screen s) -> [ window.display (s) ; content.display (s) ] ; scrollbar.display (s)
```

Aucune relation d'ordre n'est définie entre les deux messages `title.display` et `scrollbar.display`, ni entre les deux messages `content.display` et `border.display`. Par conséquent, le résultat de la fusion comportementale appliquée à ces deux règles d'interaction ne doit pas en définir une.

Ainsi, après application de la fonction de fusion comportementale, ces deux messages doivent être exécutés concurrentiellement. Ceci peut être réalisé en remplaçant (par application de la règle de transformation T1) un comportement réactif séquentiel par un comportement réactif concurrentiel.

Ainsi, le résultat de la fusion comportementale des deux règles d'interaction définies ci-dessus est la règle d'interaction suivante :

```
window.display (screen s), mailbox mb -> [ windows.display (s) ; [ [ content.display (s) ; scrollbar.display (s) ] //
                                           mb := title.display (s) ] ] // wait border.display (s) for mb
```

Ainsi, la sémantique de la fusion comportementale de deux comportements réactifs séquentiels est décrite par la règle de sémantique naturelle suivante :

$$\frac{T1\,(m_1 \rightarrow \text{sequential}\,(cr_1, cr_2)) : R}{(m_1 \rightarrow \text{sequential}\,(cr_1, cr_2)\,, m_2 \rightarrow \text{sequential}\,(cr_3, cr_4)) : (R, m_2 \rightarrow \text{sequential}\,(cr_3, cr_4))}$$

où $T1$ est l'application de la règle de transformation T1 définie en 6.2.

□

Règle de fusion R9 : Fusion d'un comportement réactif conditionnel

SÉMANTIQUE INFORMELLE. — Un comportement réactif conditionnel « encapsule » les autres comportements réactifs.

SOLUTION INTUITIVE. — Soit les deux règles d'interaction suivantes :

```
R1: math.calculus (int op, floatVect vars) -> if ctrl.hasSideEffect (op)
                                                then
                                                  math.calculus (op, vars) ; debug.mathop (op, vars)
                                                else
                                                  math.calculus (op, vars) // debug.mathop (op, vars) endif
R2: math.calculus (int op, floatVect vars) -> math.calculus (op, vars) ; ctrl.update (op)
```

Intuitivement, la fusion de ces deux règles d'interaction implique de fusionner la seconde règle d'interaction avec, d'une part, la partie `then` de la première règle d'interaction et avec, d'autre part, la partie `else` de cette même règle d'interaction. Ainsi, les deux règles d'interaction suivantes doivent être fusionnées (partie `then`) :

```
R1′: math.calculus (int op, floatVect vars) -> math.calculus (op, vars) ; debug.mathop (op, vars)
R2: math.calculus (int op, floatVect vars) -> math.calculus (op, vars) ; ctrl.update (op)
```

Le résultat de cette fusion est la règle d'interaction suivante :

```
math.calculus (int op, floatVect vars), mailbox mb -> [ mb := math.calculus (op, vars) ; ctrl.update (op) ] //
                                                        wait debug.mathop (op, vars) for mb
```

3. Nous notons $m \in cr$ l'appartenance du message m dans le comportement réactif cr.

Les deux règles d'interaction suivantes doivent, elles aussi, être fusionnées (partie*else*) :

```
R1″: math.calculus (int op, floatVect vars) -> math.calculus (op, vars) // debug.mathop (op, vars)
R2: math.calculus (int op, floatVect vars) -> math.calculus (op, vars) ; ctrl.update (op)
```

Le résultat de leur fusion comportementale est la règle d'interaction suivante :

```
math.calculus (int op, floatVect vars) -> [ math.calculus (op, vars) ; ctrl.update (op) ] //
                                           debug.mathop (op, vars)
```

De tout ceci, on peut en déduire que le résultat de la fusion comportementale de R1 et R2 est la règle d'interaction suivante :

```
math.calculus (int op, floatVect vars), mailbox mb -> if ctrl.hasSideEffect (op) then
                                                        [ mb := math.calculus (op, vars) ; ctrl.update (op) ] //
                                                          wait debug.mathop (op, vars) for mb
                                                      else
                                                        [ math.calculus (op, vars) ; ctrl.update (op) ] //
                                                          debug.mathop (op, vars) endif
```

Ainsi, la sémantique de la fusion comportementale d'un comportement réactif conditionnel avec un autre comportement réactif est décrite par les deux règles de sémantique naturelle suivantes :

$$\frac{(m_1 \to cr_1, m_2 \to cr_3) : m \to cr_{1.3} \quad (m_1 \to cr_2, m_2 \to cr_3) : m \to cr_{2.3}}{(m_1 \to \text{ifThenElse}\,(c, cr_1, cr_2), m_2 \to cr_3) : m \to \text{ifThenElse}\,(s\,(c), cr_{1.3}, cr_{2.3})}$$

$$\frac{(m_1 \to cr_1, m_2 \to cr_2) : m \to cr_{1.2}}{(m_1 \to \text{ifThen}\,(c, cr_1), m_2 \to cr_2) : m \to \text{ifThenElse}\,(s\,(c), cr_{1.2}, cr_2)}$$

□

Règle de fusion R10 : Fusion d'un comportement réactif de traitement des exceptions

SÉMANTIQUE INFORMELLE. — Un comportement réactif de traitement des exceptions est un « bloc » indivisible.

SOLUTION INTUITIVE. — Soit les deux règles d'interaction suivantes :

```
R1: master.request (string text) -> try master.request (out text) catch ( NetworkException ) slave.request (out text)
R2: master.request (string text) -> try master.request (out text) ; observer.update (text)
                                    catch ( InvalidOperation ) Reaction
```

Intuitivement, le résultat de la fusion comportementale de ces deux règles d'interaction est la règle d'interaction suivante :

```
master.request (string text) -> try [ try master.request (out text)
                                        catch ( NetworkException ) slave.request (out text) ;
                                      observer.update (text) ]
                                catch ( InvalidOperation ) Reaction
```

Ainsi, la sémantique de la fusion comportementale de deux comportements réactifs de traitements des exceptions est décrite par les trois règles de sémantique naturelle suivantes :

$$\frac{(m_1 \to cr_1, m_2 \to cr_3) : m \to cr_{1.3} \quad (m_1 \to cr_2, m_2 \to cr_4) : m \to cr_{2.4}}{(m_1 \to \text{try}\,(cr_1, e, cr_2), m_2 \to \text{try}\,(cr_3, e, cr_4)) : m \to \text{try}\,(cr_{1.3}, e, cr_{2.4})}$$

$$\frac{(m_1 \to cr_1, m_2 \to \text{try}\,(cr_3, e_2, cr_4)) : m \to cr_{1.3.4}}{\begin{array}{l}(m_1 \to \text{try}\,(cr_1, e_1, cr_2), m_2 \to \text{try}\,(cr_3, e_2, cr_4)) : \\ \qquad m \to \text{try}\,(m \to cr_{1.3.4}, e_1, s\,(cr_2)) \mid cr_3 \in \mathcal{G}_M \cup \mathcal{G}_A \cup \mathcal{G}_W\end{array}}$$

$$\frac{(m_1 \to cr_1, m_2 \to cr_3) : m \to cr_{1.3}}{(m_1 \to \text{try}\,(cr_1, e_1, cr_2), m_2 \to \text{try}\,(cr_3, e_2, cr_4)) : m \to \text{try}\,(\text{try}\,(cr_{1.3}, e_2, s\,(cr_4)), e_1, s\,(cr_2))}$$

La sémantique de la fusion comportementale d'un comportement réactif de traitement des exceptions avec un comportement réactif séquentiel est décrite par les deux règles de sémantique naturelle suivantes :

$$\frac{(m_1 \to cr_1, m_2 \to \text{try}\,(cr_3, e, cr_4)) : R}{(m_1 \to \text{sequential}\,(cr_1, cr_2), m_2 \to \text{try}\,(cr_3, e, cr_4)) : m \to \text{sequential}\,(R, s\,(cr_2)) \mid m \in cr_1}$$

$$\frac{(m_1 \to cr_2, m_2 \to \text{try}\,(cr_3, e, cr_4)) : R}{(m_1 \to \text{sequential}\,(cr_1, cr_2), m_2 \to \text{try}\,(cr_3, e, cr_4)) : m \to \text{sequential}\,(s\,(cr_1), R) \mid m \in cr_2}$$

□

6.4.3 Complétude des règles de la fusion comportementale

Le tableau 6.1 présente la règle de fusion comportementale à appliquer en fonction du type des comportements réactifs et précise le type de comportement réactif de la règle d'interaction résultante de la fusion comportementale. Ce tableau montre également la fermeture du système de fusion.

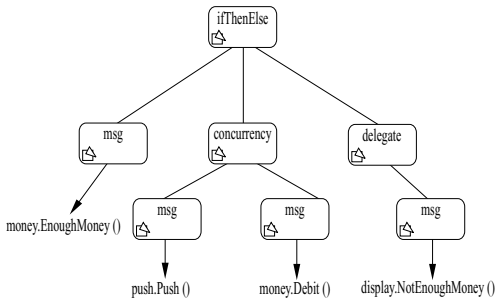
	delegate	sequential	concurrency	ifThenElse	msg	assignment	waiting	try	exception
delegate	$\frac{R1}{échec}$	$\frac{R3}{\text{sequential}}$	$\frac{R3}{\text{concurrency}}$	$\frac{R3}{\text{ifThenElse}}$	$\frac{R3}{\text{delegate}}$	$\frac{R3}{\text{assignment}}$	$\frac{R3}{\text{waiting}}$	$\frac{R3}{\text{try}}$	$\frac{R2}{\text{exception}}$
sequential	$\frac{R3}{\text{sequential}}$	$\frac{T1,R8}{\text{concurrency}}$	$\frac{R7}{\text{concurrency}}$	$\frac{R9}{\text{ifThenElse}}$	$\frac{R4}{\text{sequential}}$	$\frac{R5}{\text{sequential}}$	$\frac{R6}{\text{sequential}}$	$\frac{R10}{\text{sequential}}$	$\frac{R2}{\text{exception}}$
concurrency	$\frac{R3}{\text{concurrency}}$	$\frac{R7}{\text{concurrency}}$	$\frac{R7}{\text{concurrency}}$	$\frac{R7}{\text{concurrency}}$	$\frac{R4}{\text{concurrency}}$	$\frac{R7}{\text{concurrency}}$	$\frac{R7}{\text{concurrency}}$	$\frac{R7}{\text{concurrency}}$	$\frac{R2}{\text{exception}}$
ifThenElse	$\frac{R3}{\text{ifThenElse}}$	$\frac{R9}{\text{ifThenElse}}$	$\frac{R7}{\text{concurrency}}$	$\frac{R9}{\text{ifThenElse}}$	$\frac{R4}{\text{ifThenElse}}$	$\frac{R9}{\text{ifThenElse}}$	$\frac{R9}{\text{ifThenElse}}$	$\frac{R9}{\text{ifThenElse}}$	$\frac{R2}{\text{exception}}$
msg	$\frac{R3}{\text{delegate}}$	$\frac{R4}{\text{sequential}}$	$\frac{R4}{\text{concurrency}}$	$\frac{R4}{\text{ifThenElse}}$	$\frac{R4}{\text{msg}}$	$\frac{R4}{\text{assignment}}$	$\frac{R4}{\text{waiting}}$	$\frac{R4}{\text{try}}$	$\frac{R2}{\text{exception}}$
assignment	$\frac{R3}{\text{assignment}}$	$\frac{R5}{\text{sequential}}$	$\frac{R7}{\text{concurrency}}$	$\frac{R9}{\text{ifThenElse}}$	$\frac{R4}{\text{assignment}}$	$\frac{R5}{\text{assignment}}$	$\frac{R6}{\text{waiting}}$	$\frac{R5}{\text{try}}$	$\frac{R2}{\text{exception}}$
waiting	$\frac{R3}{\text{waiting}}$	$\frac{R6}{\text{sequential}}$	$\frac{R7}{\text{concurrency}}$	$\frac{R9}{\text{ifThenElse}}$	$\frac{R4}{\text{waiting}}$	$\frac{R6}{\text{waiting}}$	$\frac{R6}{\text{waiting}}$	$\frac{R6}{\text{try}}$	$\frac{R2}{\text{exception}}$
try	$\frac{R3}{\text{try}}$	$\frac{R10}{\text{seqential}}$	$\frac{R7}{\text{concurrency}}$	$\frac{R9}{\text{ifThenElse}}$	$\frac{R4}{\text{try}}$	$\frac{R5}{\text{try}}$	$\frac{R6}{\text{try}}$	$\frac{R10}{\text{try}}$	$\frac{R2}{\text{exception}}$
exception	$\frac{R2}{\text{exception}}$	$\frac{R2}{\text{exception}}$	$\frac{R2}{\text{exception}}$	$\frac{R2}{\text{exception}}$	$\frac{R2}{\text{exception}}$	$\frac{R2}{\text{exception}}$	$\frac{R2}{\text{exception}}$	$\frac{R2}{\text{exception}}$	$\frac{R2}{\text{exception}}$

TAB. 6.1 – *Complétude des règles de la fusion comportementale*

6.5 Propriété de commutativité de la fusion comportementale

Propriété 6.2 : Commutativité de la fonction de fusion comportementale
La fonction de fusion comportementale est commutative, c'est-à-dire qu'elle vérifie la propriété suivante (on rappelle que deux règles d'interaction sont équivalentes si et seulement si la sémantique de leur exécution est identique) :

$$\forall (r_1, r_2) \in \mathcal{R}^2 \quad \varphi(r_1, r_2) \equiv \varphi(r_2, r_1)$$



Pour cette preuve, nous allons supposer que les comportements réactifs décrits par les règles d'interaction sont représentés par des arbres (figure ci-contre) dont les noeuds sont les opérateurs réactifs non terminaux (sequential ou ifThenElse par exemple), et les feuilles les opérateurs terminaux (msg ou assign par exemple).

L'arbre de syntaxe abstraite du langage ISL est une représentation possible.

Nous allons prouver cette propriété par récurrence sur la hauteur des arbres de syntaxe abstraite décrivant les règles d'interaction à fusionner. Nous avons donc l'hypothèse de récurrence suivante [4] :

$$\forall n \in \mathbb{N}^*, \forall (r_1, r_2, r_3) \in \mathcal{R}^3 \mid \hbar(r_1) = n-1 \wedge \hbar(r_3) = n \wedge r_1 \in r_3$$
$$\varphi(r1, r2) \equiv \varphi(r_2, r_1) \Rightarrow \varphi(r3, r2) \equiv \varphi(r_2, r_3)$$

4. Nous notons $r_1 \in r_2$ le fait que l'arbre de r_1 est un sous-arbre de r_2. De même nous notons $\hbar$ la fonction associant à une règle d'interaction la hauteur de son arbre de syntaxe abstraite. Son domaine de définition est : $\hbar : \mathcal{R} \rightarrow \mathbb{N}$

De cette hypothèse de récurrence, nous pouvons en déduire l'hypothèse de récurrence étendue suivante :

$$\forall n \in \mathbb{N}^*, \forall (r_1, r_2, r_3) \in \mathcal{R}^3 \mid \hbar(r_1) \leqslant n-1 \wedge \hbar(r_3) = n \wedge r_1 \in r_3$$
$$\varphi(r_1, r_2) \equiv \varphi(r_2, r_1) \Rightarrow \varphi(r_3, r_2) \equiv \varphi(r_2, r_3)$$

6.5.1 Preuve pour les axiomes (règles terminales) de la fusion comportementale

Preuve de la commutativité de la règle de fusion des exceptions (R2) :
Nous avons les deux axiomes suivants :

$$(R1 : m_1 \rightarrow \text{exception}(e_1), R2 : m_2 \rightarrow \text{exception}(e_2)) : m \rightarrow \text{concurrency}(\text{exception}(e_1), \text{exception}(e_2))$$

$$(m_1 \rightarrow cr_1, m_2 \rightarrow \text{exception}(e)) : m \rightarrow \text{exception}(e)$$

Dans le cas du premier axiome, nous avons les équivalences suivantes :

$$\begin{aligned} \varphi(R_1, R_2) &= m \rightarrow \text{concurrency}(\text{exception}(e_1), \text{exception}(e_2)) \\ &\equiv m \rightarrow \text{concurrency}(\text{exception}(e_2), \text{exception}(e_1)) \qquad \text{(par application de T4)} \\ &\equiv \varphi(R_2, R_1) \end{aligned}$$

Le second axiome stipule qu'un comportement réactif d'exception est absorbant vis-à-vis des autres comportements réactifs. Ainsi, par définition, ce second axiome indique que la fusion d'un comportement réactif d'exception avec un autre comportement réactif est commutative.

Ainsi la règle de fusion comportementale R2 est commutative.

◇

Preuve de la commutativité de la règle de fusion de délégation (R3) :
La commutativité de cette règle de fusion comportementale est vérifiée par définition.

◇

Preuve de la commutativité de la règle de fusion d'envoi de messages (R4) :
Nous avons l'axiome suivant :

$$(m_1 \rightarrow \text{msg}(o, m), m_2 \rightarrow cr) : m \rightarrow s(cr)$$

Cet axiome stipule qu'un comportement réactif d'envoi de messages est neutre vis-à-vis des autres comportements réactifs. Ainsi, par définition, cet axiome indique que la fusion d'un comportement réactif d'envoi de messages avec un autre comportement réactif est commutative.

◇

Nous venons de prouver que la propriété de commutativité de la fonction de fusion comportementale est vérifiée pour chacun des axiomes. Nous pouvons maintenant utiliser l'hypothèse de récurrence étendue et montrer la propriété de commutativité pour les autres règles de fusion comportementale.

6.5.2 Preuve pour les règles non terminales de la fusion comportementale

NOTE. — La preuve de la commutativité de la fonction de fusion comportementale, pour chacune des règles de fusion comportementale non terminale, est basée sur le même schéma de preuve. Celui-ci consiste à utiliser l'hypothèse de récurrence ainsi que les règles d'équivalence (présentées au paragraphe 6.2) afin d'obtenir du résultat de la fusion de deux règles d'interaction r_1 avec r_2 le résultat de la fusion de r_2 avec r_1.

Par conséquent, nous ne détaillons ce schéma de preuve que pour la première règle de fusion comportementale (R5) et, pour les autres règles de fusion, ne présentons que les étapes essentielles à la compréhension de la preuve.

Preuve de la commutativité de la règle de fusion d'affectation (R5) :
La fusion de deux comportements réactifs d'affectation est décrite par la règle de sémantique naturelle suivante :

$$\frac{(m_1 \rightarrow cr_1, m_2 \rightarrow cr_2) : m \rightarrow cr_{1.2}}{(R1 : m_1 \rightarrow \text{assignment}(x_1, cr_1), R2 : m_2 \rightarrow \text{assignment}(x_2, cr_2)) : m \rightarrow \text{assignment}(x_1, \text{assignment}(x_2, cr_{1.2}))}$$

Grâce à l'hypothèse de récurrence, nous avons [5] :

$$\varphi\left(m_1 \to cr_1, m_2 \to cr_2\right) = m \to cr_{1.2} \equiv \varphi\left(m_2 \to cr_2, m_1 \to cr_1\right) = m \to cr_{2.1}$$

De ce fait, nous pouvons en déduire que :

$$\begin{array}{rcll} \varphi\left(R_1, R_2\right) & = & m \to \text{assignment}\left(x_1, \text{assignment}\left(x_2, cr_{1.2}\right)\right) & \\ & \equiv & m \to \text{assignment}\left(x_1, \text{assignment}\left(x_2, cr_{2.1}\right)\right) & \text{(par hypothèse de récurrence)} \\ & \equiv & m \to \text{assignment}\left(x_2, \text{assignment}\left(x_1, cr_{2.1}\right)\right) & \text{(par application de T4)} \\ & \equiv & \varphi\left(R_2, R_1\right) & \end{array}$$

La fusion comportementale d'un comportement réactif d'affectation avec un comportement réactif séquentiel est décrite par deux règles de sémantique naturelle. La preuve de commutativité est en tout point identique pour chacune de ces deux règles. Par conséquent, nous ne détaillons ici que la preuve de commutativité pour la première règle, à savoir :

$$\frac{\left(m_1 \to cr_1, m_2 \to \text{assignment}\left(x, cr_3\right)\right) : m \to cr_{1.3}}{\left(R1 : m_1 \to \text{sequential}\left(cr_1, cr_2\right), R2 : m_2 \to \text{assignment}\left(x, cr_3\right)\right) : m \to \text{sequential}\left(cr_{1.3}, s\left(cr_2\right)\right) \mid m \in cr_1}$$

Grâce à l'hypothèse de récurrence, nous avons :

$$\varphi\left(m_1 \to cr_1, m_2 \to \text{assignment}\left(x, cr_3\right)\right) = m \to cr_{1.3} \equiv \varphi\left(m_2 \to \text{assignment}\left(x, cr_3\right), m_1 \to cr_1\right) = m \to cr_{3.1}$$

De ce fait, nous pouvons en déduire que :

$$\begin{array}{rcll} \varphi\left(R_1, R_2\right) & = & m \to \text{sequential}\left(cr_{1.3}, s\left(cr_2\right)\right) & \\ & \equiv & m \to \text{sequential}\left(cr_{3.1}, s\left(cr_2\right)\right) & \text{(par hypothèse de récurrence)} \\ & \equiv & \varphi\left(R_2, R_1\right) & \end{array}$$

La fusion comportementale d'un comportement réactif d'affectation avec un comportement réactif de traitement des exceptions est décrite par la règle de sémantique naturelle suivante :

$$\frac{\left(m_1 \to cr_1, m_2 \to \text{assignment}\left(x, cr_3\right)\right) : m \to cr_{1.3}}{\left(R1 : m_1 \to \text{try}\left(cr_1, e, cr_2\right), R2 : m_2 \to \text{assignment}\left(x, cr_3\right)\right) : m \to \text{try}\left(cr_{1.3}, e, s\left(cr_2\right)\right)}$$

Grâce à l'hypothèse de récurrence définie ci-dessus nous pouvons en déduire que :

$$\begin{array}{rcll} \varphi\left(R_1, R_2\right) & = & m \to \text{try}\left(cr_{1.3}, e, s\left(cr_2\right)\right) & \\ & \equiv & m \to \text{try}\left(cr_{3.1}, e, s\left(cr_2\right)\right) & \text{(par hypothèse de récurrence)} \\ & \equiv & \varphi\left(R_2, R_1\right) & \end{array}$$

Ainsi, la règle de fusion comportementale R5 est commutative.

◇

Preuve de la commutativité de la règle de fusion d'attente (R6) :
La fusion comportementale d'un comportement réactif d'attente avec un comportement réactif d'affectation est décrite par la règle de sémantique naturelle suivante :

$$\frac{\left(m_1 \to cr_1, m_2 \to \text{assignment}\left(x, cr_2\right)\right) : m \to cr_{1.2}}{\left(m_1 \to \text{waiting}\left(cr_1, mb\right), m_2 \to \text{assignment}\left(x, cr_2\right)\right) : m \to \text{waiting}\left(cr_{1.2}, mb\right)}$$

Grâce à l'hypothèse de récurrence, nous avons :

$$\varphi\left(m_1 \to cr_1, m_2 \to \text{assignment}\left(x, cr_2\right)\right) = m \to cr_{1.2} \equiv \varphi\left(m_2 \to \text{assignment}\left(x, cr_2\right), m_1 \to cr_1\right) = m \to cr_{2.1}$$

De ce fait, nous pouvons en déduire que :

$$\begin{array}{rcll} \varphi\left(R_1, R_2\right) & = & m \to \text{waiting}\left(cr_{1.2}, mb\right) & \\ & \equiv & m \to \text{waiting}\left(cr_{2.1}, mb\right) & \text{(par hypothèse de récurrence)} \\ & \equiv & \varphi\left(R_2, R_1\right) & \end{array}$$

Les preuves de la propriété de commutativité de la fonction de fusion comportementale concernant les autres règles de sémantique naturelle de la règle de fusion R6 sont en tout point identique aux preuves de la commutativité de la fonction de fusion comportementale présentées pour la règle de fusion R5. Nous ne les détaillons donc pas.

Ainsi, la règle de fusion comportementale R6 est commutative.

◇

5. Nous notons $cr_{x.y}$ le comportement réactif résultant de la fusion comportementale du comportement réactif cr_x avec le comportement réactif cr_y.

Preuve de la commutativité de la règle de fusion d'un comportement réactif concurrentiel (R7) :
La fusion comportementale d'un comportement réactif concurrentiel avec un autre comportement réactif est décrite par la règle de sémantique naturelle suivante :

$$\frac{(m_1 \to cr_2, m_2 \to cr_3) : m \to cr_{2.3}}{(R1 : m_1 \to \text{concurrency}\,(cr_1, cr_2), R2 : m_2 \to cr_3) : m \to \text{concurrency}\,(s\,(cr_1), cr_{2.3}) \mid m_1 \in cr_2}$$

Grâce à l'hypothèse de récurrence, nous avons :

$$\varphi\,(m_1 \to cr_2, m_2 \to cr_3) = m \to cr_{2.3} \equiv \varphi\,(m_2 \to cr_3, m_1 \to cr_2) = m \to cr_{3.2}$$

De ce fait, nous pouvons en déduire que :

$$\begin{aligned} \varphi\,(R_1, R_2) &= m \to \text{concurrency}\,(s\,(cr_1), cr_{2.3}) \\ &\equiv m \to \text{concurrency}\,(s\,(cr_1), cr_{3.2}) \\ &\equiv \varphi\,(R_2, R_1) \end{aligned}$$

Ainsi la règle de fusion comportementale R7 est commutative.

◇

Preuve de la commutativité de la règle de fusion d'un comportement réactif séquentiel (R8) :
La fusion comportementale de deux comportements réactifs séquentiels est définie par la fusion comportementale de l'un des deux comportements réactifs séquentiels avec la transformation de l'autre comportement réactif séquentiel en un comportement réactif concurrentiel. Or la fusion comportementale d'un comportement réactif concurrentiel avec un autre comportement réactif est commutative (preuve ci-dessus) donc la fusion comportementale de deux comportements réactifs séquentiels est commutative.

Ainsi la règle de fusion comportementale R8 est commutative.

◇

Preuve de la commutativité de la règle de fusion d'un comportement réactif conditionnel (R9) :
La fusion comportementale d'un comportement réactif conditionnel avec un autre comportement réactif est notamment décrite par la règle de sémantique naturelle suivante :

$$\frac{\begin{array}{l}(m_1 \to cr_1, m_2 \to cr_3) : m \to cr_{1.3} \\ (m_1 \to cr_2, m_2 \to cr_3) : m \to cr_{2.3}\end{array}}{(R1 : m_1 \to \text{ifThenElse}\,(c, cr_1, cr_2), R2 : m_2 \to cr_3) : m \to \text{ifThenElse}\,(s\,(c), cr_{1.3}, cr_{2.3})}$$

Grâce à l'hypothèse de récurrence, nous avons :

$$\begin{aligned} \varphi\,(m_1 \to cr_1, m_2 \to cr_3) = m \to cr_{1.3} &\equiv \varphi\,(m_2 \to cr_3, m_1 \to cr_1) = m \to cr_{3.1} \\ \varphi\,(m_1 \to cr_2, m_2 \to cr_3) = m \to cr_{2.3} &\equiv \varphi\,(m_2 \to cr_3, m_1 \to cr_2) = m \to cr_{3.2} \end{aligned}$$

De ce fait, nous pouvons en déduire que :

$$\begin{aligned} \varphi\,(R_1, R_2) &= m \to \text{ifThenElse}\,(s\,(c), cr_{1.3}, cr_{2.3}) \\ &\equiv m \to \text{ifThenElse}\,(s\,(c), cr_{3.1}, cr_{3.2}) \\ &\equiv \varphi\,(R_2, R_1) \end{aligned}$$

La preuve de la propriété de commutativité de la fonction de fusion comportementale concernant la seconde règle de sémantique naturelle de la règle de fusion R9 est en tout point identique. Nous ne la détaillons donc pas.

Ainsi la règle de fusion comportementale R9 est commutative.

◇

Preuve de la commutativité de la règle de fusion d'un comportement réactif de traitement des exceptions (R10) :

La fusion comportementale de deux comportements réactifs de traitement de la même exception est décrite par la règle de sémantique naturelle suivante :

$$\frac{\begin{array}{l}(m_1 \to cr_1, m_2 \to cr_3) : m \to cr_{1.3} \\ (m_1 \to cr_2, m_2 \to cr_4) : m \to cr_{2.4}\end{array}}{(m_1 \to \text{try}\,(cr_1, e, cr_2), m_2 \to \text{try}\,(cr_3, e, cr_4)) : m \to \text{try}\,(cr_{1.3}, e, cr_{2.4})}$$

Grâce à l'hypothèse de récurrence, nous avons :

$$\begin{aligned} \varphi\,(m_1 \to cr_1, m_2 \to cr_3) = m \to cr_{1.3} &\equiv \varphi\,(m_2 \to cr_3, m_1 \to cr_1) = m \to cr_{3.1} \\ \varphi\,(m_1 \to cr_2, m_2 \to cr_4) = m \to cr_{2.4} &\equiv \varphi\,(m_2 \to cr_4, m_1 \to cr_2) = m \to cr_{4.2} \end{aligned}$$

De ce fait, nous pouvons en déduire que :

$$\begin{aligned}\varphi(R_1,R_2) &= m \to \text{try}(cr_{1.3},e,cr_{2.4})\\ &\equiv m \to \text{try}(cr_{3.1},e,cr_{4.2})\\ &\equiv \varphi(R_2,R_1)\end{aligned}$$

La fusion comportementale de deux comportements réactifs de traitement de deux exceptions différentes est décrite par les deux règles de sémantique naturelle suivantes :

$$\frac{(m_1 \to cr_1, m_2 \to \text{try}(cr_3,e_2,cr_4)) : m \to cr_{1.3.4}}{\begin{array}{l}(R1 : m_1 \to \text{try}(cr_1,e_1,cr_2), R2 : m_2 \to \text{try}(cr_3,e_2,cr_4)) :\\ \qquad m \to \text{try}(m \to cr_{1.3.4}, e_1, s(cr_2)) \mid cr_3 \in \mathcal{G}_M \cup \mathcal{G}_A \cup \mathcal{G}_W\end{array}}$$

$$\frac{(m_1 \to cr_1, m_2 \to cr_3) : m \to cr_{1.3}}{(R3 : m_1 \to \text{try}(cr_1,e_1,cr_2), R4 : m_2 \to \text{try}(cr_3,e_2,cr_4)) : m \to \text{try}(\text{try}(cr_{1.3},e_2,s(cr_4)),e_1,s(cr_2))}$$

Grâce à l'hypothèse de récurrence, nous avons :

$$\varphi(m_1 \to cr_1, m_2 \to \text{try}(cr_3,e_2,cr_4)) = m \to cr_{1.3.4} \equiv \varphi(m_2 \to \text{try}(cr_3,e_2,cr_4), m_1 \to cr_1) = m \to cr_{3.4.1}$$
$$\varphi(m_1 \to cr_1, m_2 \to cr_3) = m \to cr_{1.3} \equiv \varphi(m_2 \to cr_3, m_1 \to cr_1) = m \to cr_{3.1}$$

De ce fait, pour la première règle de sémantique naturelle, nous pouvons en déduire que :

$$\begin{aligned}\varphi(R_1,R_2) &= m \to \text{try}(cr_{1.3.4},e_1,s(cr_2))\\ &\equiv m \to \text{try}(cr_{3.4.1},e_1,s(cr_2))\\ &\equiv \varphi(R_2,R_1)\end{aligned}$$

De même, pour la seconde règle de sémantique naturelle, nous pouvons en déduire que :

$$\begin{aligned}\varphi(R_3,R_4) &= m \to \text{try}(\text{try}(cr_{1.3},e_2,s(cr_4)),e_1,s(cr_2))\\ &\equiv m \to \text{try}(\text{try}(cr_{3.1},e_2,s(cr_4)),e_1,s(cr_2))\\ &\equiv m \to \text{try}(\text{try}(cr_{3.1},e_1,s(cr_2)),e_2,s(cr_4)) \qquad \text{(par application de T4)}\\ &\equiv \varphi(R_4,R_3)\end{aligned}$$

Les preuves de la propriété de commutativité de la fonction de fusion comportementale concernant les autres règles de sémantique naturelle de la règle de fusion R10 sont en tout point identique aux preuves de la commutativité de la fonction de fusion comportementales présentées ci-dessus. Nous ne les détaillons donc pas.

Ainsi la règle de fusion comportementale R10 est commutative.

◇

Nous venons par conséquent de vérifier que toutes les règles de fusion comportementale vérifiaient la propriété de commutativité :

$$\forall (r_1,r_2) \in \mathcal{R}^2 \quad \varphi(r_1,r_2) \equiv \varphi(r_2,r_1)$$

Ainsi, la fonction de fusion comportementale est commutative.

□

6.6 Propriété d'associativité de la fusion comportementale

Propriété 6.3 : Associativité de la fonction de fusion comportementale
La fonction de fusion comportementale est associative, c'est-à-dire qu'elle vérifie la propriété suivante :

$$\forall (r_1,r_2,r_3) \in \mathcal{R}^3 \quad \varphi(\varphi(r_1,r_2),r_3) \equiv \varphi(r_1,\varphi(r_2,r_3))$$

Tout comme pour prouver la commutativité de la fonction de fusion comportementale, nous allons prouver cette propriété d'associativité par récurrence sur la hauteur des arbres de syntaxe abstraite décrivant les règles d'interaction à fusionner. Nous avons donc l'hypothèse de récurrence suivante :

$$\begin{array}{l}\forall n \in \mathbb{N}^*, \forall (r_1,r_2,r_3,r_4) \in \mathcal{R}^4 \mid \hbar(r_1) = n-1 \wedge \hbar(r_3) = n \wedge r_1 \in r_3\\ \varphi(\varphi(r_4,r_2),r_3) \equiv \varphi(r_4,\varphi(r_2,r_3)) \Rightarrow \varphi(\varphi(r_1,r_2),r_3) \equiv \varphi(r_1,\varphi(r_2,r_3))\end{array}$$

De cette hypothèse de récurrence nous pouvons en déduire l'hypothèse de récurrence étendue suivante :

$$\begin{array}{l}\forall n \in \mathbb{N}^*, \forall (r_1,r_2,r_3,r_4) \in \mathcal{R}^4 \mid \hbar(r_1) \leqslant n-1 \wedge \hbar(r_3) = n \wedge r_1 \in r_3\\ \varphi(\varphi(r_4,r_2),r_3) \equiv \varphi(r_4,\varphi(r_2,r_3)) \Rightarrow \varphi(\varphi(r_1,r_2),r_3) \equiv \varphi(r_1,\varphi(r_2,r_3))\end{array}$$

6.6.1 Preuve pour les axiomes (règles terminales) de la fusion comportementale

Preuve de l'associativité de la règle de fusion des exceptions (R2) :
Nous avons les deux axiomes suivants :

$$(m_1 \rightarrow \text{exception}(e_1), m_2 \rightarrow \text{exception}(e_2)) : m \rightarrow \text{concurrency}(\text{exception}(e_1), \text{exception}(e_2))$$

$$(m_1 \rightarrow cr_1, m_2 \rightarrow \text{exception}(e)) : m \rightarrow \text{exception}(e)$$

Nous pouvons en déduire que :

- $R_1 \in \mathcal{G}_E,\ (R_2, R_3) \in \mathcal{G}^2_{\neg E}$

$$\begin{aligned} \varphi(R_1, \varphi(R_2, R_3)) &= R_1 \qquad \text{par application du second axiome de la règle R2} \\ \text{et } \varphi(\varphi(R_1, R_2), R_3) &= \varphi(R_1, R_3) \\ &= R_1 \end{aligned}$$

- $(R_1, R_2) \in \mathcal{G}^2_E,\ R_3 \in \mathcal{G}_{\neg E}$

$$\begin{aligned} \varphi(R_1, \varphi(R_2, R_3)) &= \underbrace{\varphi(R_1, R_2)}_{\in \mathcal{G}_E} \text{ car } \varphi(R_2, R_3) = R_2 \\ &= \varphi(\varphi(R_1, R_2), R_3) \text{ puisque } R_3 \in \mathcal{G}_{\neg E} \end{aligned}$$

- $R_1 \in \mathcal{G}_E : m_1 \rightarrow \text{exception}(e_1), \quad R_2 \in \mathcal{G}_E : m_2 \rightarrow \text{exception}(e_2), \quad R_3 \in \mathcal{G}_E : m_3 \rightarrow \text{exception}(e_3)$

$$\begin{aligned} \varphi(\varphi(R_1, R_2), R_3) &= \varphi(m \rightarrow \text{concurrency}(\text{exception}(e_1), \text{exception}(e_2)), R_3) \\ &= m \rightarrow \text{concurrency}(\text{concurrency}(\text{exception}(e_1), \text{exception}(e_2)), \text{exception}(e_3)) \\ \text{et } \varphi(R_1, \varphi(R_2, R_3)) &= \varphi(R_1, m \rightarrow \text{concurrency}(\text{exception}(e_2), \text{exception}(s(e_3)))) \\ &= m \rightarrow \text{concurrency}(\text{exception}(e_1), \text{concurrency}(\text{exception}(e_2), \text{exception}(e_3))) \\ &= \varphi(\varphi(R_1, R_2), R_3) \end{aligned}$$

Ainsi la règle de fusion comportementale R2 est associative.

◇

Preuve de l'associativité de la règle de fusion de délégation (R3) :
La fusion comportementale d'un comportement réactif de délégation avec un autre comportement réactif est notamment décrite par les deux axiomes de sémantique naturelle suivants :

$$(m_1 \rightarrow \text{delegate}(cr_1), m_2 \rightarrow cr_2) : m \rightarrow \text{subst}(m, s(\text{delegate}(cr_1)), s(cr_2))$$

$$(m_1 \rightarrow cr_1, m_2 \rightarrow \text{delegate}(cr_2)) : m \rightarrow \text{subst}(m, s(\text{delegate}(cr_2)), s(cr_1))$$

où subst est une fonction de substitution.

Soit $R_1 : m_1 \rightarrow \text{delegate}(cr_1)$, $R_2 : m_2 \rightarrow cr_2$, et $R_3 : m_3 \rightarrow cr_3$.

Nous pouvons en déduire que :

$$\begin{aligned} \varphi(R_1, \varphi(R_2, R_3)) &\equiv \varphi(m_1 \rightarrow \text{delegate}(cr_1), \varphi(R_2, R_3)) \\ &\equiv m \rightarrow \mathit{subst}(m, s(\text{delegate}(cr_1)), \varphi(R_2, R_3)) \\ \text{et } \varphi(\varphi(R_1, R_2), R_3) &\equiv \varphi(m \rightarrow \mathit{subst}(s(\text{delegate}(cr_1)), s(cr_2)), R_3) \end{aligned}$$

Or la sémantique d'un comportement réactif de délégation est strictement le même que celui du message interagissant. Ainsi le résultat de la fusion de la règle R_2, à laquelle le message interagissant a été substitué par le comportement réactif de délégation de la règle R_1, avec la règle R_3 est strictement identique au résultat de la fusion des règles R_2 et R_3 auquel le message interagissant est substitué par le comportement réactif de délégation de la règle R_1.

Nous avons par conséquent l'équivalence suivante :

$$m \rightarrow \mathit{subst}(m, s(\text{delegate}(cr_1)), \varphi(R_2, R_3)) = \varphi(m \rightarrow \mathit{subst}(s(\text{delegate}(cr_1)), s(cr_2)), R_3)$$

Nous pouvons en déduire que :

$$\varphi(R_1, \varphi(R_2, R_3)) \equiv \varphi(\varphi(R_1, R_2), R_3)$$

Ainsi la règle de fusion comportementale R3 est associative.

◇

Preuve de l'associativité de la règle de fusion des envois de messages (R4) :
La fusion comportementale d'un comportement réactif d'envoi de messages avec un autre comportement réactif est décrite par la règle de sémantique naturelle suivante :

$$(m_1 \to \text{msg}(o,m), m_2 \to cr) : m \to s(cr)$$

Soit $R_1 : m_1 \to \text{msg}(o,m)$ et $(R_2, R_3) \in \mathcal{G}^2_{\neg E}$.

Nous pouvons en déduire que :

$$\begin{aligned} \varphi(\varphi(R_1,R_2),R_3) &= \varphi(\varphi(m_1 \to \text{msg}(o,m),R_2),R_3) \\ &\equiv \varphi(R_2,R_3) \\ \text{et } \varphi(R_1,\varphi(R_2,R_3)) &= \varphi(m_1 \to \text{msg}(o,m),\varphi(R_2,R_3)) \\ &\equiv \varphi(R_2,R_3) \end{aligned}$$

Ainsi la règle de fusion comportementale R4 est associative.

◇

Nous venons de prouver que la propriété d'associativité de la fonction de fusion comportementale est vérifiée pour les cas de base. Nous pouvons maintenant utiliser l'hypothèse de récurrence étendue et montrer la propriété d'associativité pour les autres règles de fusion comportementale.

6.6.2 Preuve pour les règles non terminales de la fusion comportementale

NOTE. — Les preuves de l'associativité de la fonction de fusion comportementale concernant les règles non terminales de la fusion comportementale sont toutes basées sur le même principe. De ce fait, nous le présentons en détail pour la première règle non terminale (règle R5) et ne présentons que les étapes essentielles à la compréhension des autres preuves.

Preuve de l'associativité de la règle de fusion d'affectation (R5) :
La fusion de deux comportements réactifs d'affectation est décrite par la règle de sémantique naturelle suivante :

$$\frac{(m_1 \to cr_1, m_2 \to cr_2) : m \to cr_{1.2}}{(R1 : m_1 \to \text{assignment}(x_1, cr_1), R2 : m_2 \to \text{assignment}(x_2, cr_2)) : m \to \text{assignment}(x_1, \text{assignment}(x_2, cr_{1.2}))}$$

Soit $R_1 : m_1 \to \text{assignment}(x_1, cr_1)$, $R_2 : m_2 \to \text{assignment}(x_2, cr_2)$, et $R_3 : m_3 \to \text{assignment}(x_3, cr_3)$.

Grâce à l'hypothèse de récurrence, nous avons :

$$\varphi(\varphi(m_1 \to cr_1, m_2 \to cr_2), m_3 \to cr_3) = cr_{(1.2).3} \equiv \varphi(m_1 \to cr_1, \varphi(m_2 \to cr_2, m_3 \to cr_3)) = cr_{1.(2.3)}$$

De ce fait, nous pouvons en déduire que :

$$\begin{aligned} \varphi(\varphi(R_1,R_2),R_3) &\equiv \varphi(m \to \text{assignment}(x_1, \text{assignment}(x_2, cr_{1.2})), R_3) \\ &\equiv m \to \text{assignment}\left(x_1, \text{assignment}\left(x_2, \text{assignment}\left(x_3, cr_{(1.2).3}\right)\right)\right) \\ \text{et } \varphi(R_1,\varphi(R_2,R_3)) &\equiv \varphi(R_1, m \to \text{assignment}(x_2, \text{assignment}(x_3, cr_{2.3}))) \\ &\equiv m \to \text{assignment}\left(x_1, \text{assignment}\left(x_2, \text{assignment}\left(x_3, cr_{1.(2.3)}\right)\right)\right) \\ &\equiv m \to \text{assignment}\left(x_1, \text{assignment}\left(x_2, \text{assignment}\left(x_3, cr_{(1.2).3}\right)\right)\right) \\ &\equiv \varphi(\varphi(R_1,R_2),R_3) \end{aligned}$$

La fusion comportementale d'un comportement réactif d'affectation avec un comportement réactif séquentiel est décrite par deux règles de sémantique naturelle. La preuve de l'associativité est en tout point identique pour chacune de ces deux règles. Par conséquent, nous ne détaillons ici que la preuve de l'associativité pour la première règle, à savoir :

$$\frac{(m_1 \to cr_1, m_2 \to \text{assignment}(x, cr_3)) : m \to cr_{1.3}}{(R1 : m_1 \to \text{sequential}(cr_1, cr_2), R2 : m_2 \to \text{assignment}(x, cr_3)) : m \to \text{sequential}(cr_{1.3}, s(cr_2)) \mid m \in cr_1}$$

Soit $R_1 : m_1 \to \text{sequential}(cr_1, cr_2)$, $R_2 : m_2 \to \text{assignment}(x_1, cr_3)$, et $R_3 : m_3 \to \text{assignment}(x_2, cr_4)$.

Grâce à l'hypothèse de récurrence, nous avons :

$$\varphi(\varphi(m_1 \to cr_1, R_2), R_3) = cr_{(1.2).3} \equiv \varphi(m_1 \to cr_1, \varphi(R_2, R_3)) = cr_{1.(2.3)}$$

De ce fait, nous pouvons en déduire que :

$$\begin{aligned}
\varphi(\varphi(R_1,R_2),R_3) &\equiv \varphi(m \to \text{sequential}(cr_{1.3}, s(cr_2)),R_3)\\
&\equiv m \to \text{sequential}\left(cr_{(1.2).3}, s(cr_2)\right)\\
\text{et } \varphi(R_1,\varphi(R_2,R_3)) &\equiv m \to \text{sequential}(\varphi(m_1 \to cr_1, \varphi(R_2,R_3)), s(cr_2))\\
&\equiv m \to \text{sequential}\left(cr_{(1.2).3}, s(cr_2)\right)\\
&\equiv \varphi(\varphi(R_1,R_2),R_3)
\end{aligned}$$

La fusion comportementale d'un comportement réactif d'affectation avec un comportement réactif de traitement des exceptions est décrite par la règle de sémantique naturelle suivante :

$$\frac{(m_1 \to cr_1, m_2 \to \text{assignment}(x, cr_3)) : m \to cr_{1.3}}{(R1 : m_1 \to \text{try}(cr_1, e, cr_2), R2 : m_2 \to \text{assignment}(x, cr_3)) : m \to \text{try}(cr_{1.3}, e, s(cr_2))}$$

Soit $R_1 : m_1 \to \text{try}(cr_1, e, cr_2)$, $R_2 : m_2 \to \text{assignment}(x_1, cr_3)$, $R_3 : m_3 \to \text{assignment}(x_2, cr_4)$.

Grâce à l'hypothèse de récurrence définie ci-dessus nous pouvons en déduire que :

$$\begin{aligned}
\varphi(\varphi(R_1,R_2),R_3) &\equiv m \to \text{try}(\varphi(\varphi(m_1 \to cr_1, R_2),R_3), e, s(cr_2))\\
\text{et } \varphi(R_1,\varphi(R_2,R_3)) &\equiv m \to \text{try}(\varphi(m_1 \to cr_1, \varphi(R_2,R_3)), e, s(cr_2))\\
&\equiv m \to \text{try}(\varphi(\varphi(m_1 \to cr_1, R_2),R_3), e, s(cr_2))\\
&\equiv \varphi(\varphi(R_1,R_2),R_3)
\end{aligned}$$

Ainsi la règle de fusion comportementale R5 est associative.

◇

Preuve de l'associativité de la règle de fusion d'attente (R6) :
La fusion comportementale d'un comportement réactif d'attente avec un comportement réactif d'affectation est décrite par la règle de sémantique naturelle suivante :

$$\frac{(m_1 \to cr_1, m_2 \to \text{assignment}(x, cr_2)) : m \to cr_{1.2}}{(m_1 \to \text{waiting}(cr_1, mb), m_2 \to \text{assignment}(x, cr_2)) : m \to \text{waiting}(cr_{1.2}, mb)}$$

Soit $R_1 : m_1 \to \text{waiting}(cr_1, mb)$, $R_2 : m_2 \to \text{assignment}(x_1, cr_2)$, et $R_3 : m_3 \to \text{assignment}(x_2, cr_3)$.

Grâce à l'hypothèse de récurrence, nous avons :

$$\varphi(\varphi(m_1 \to cr_1, R_2), R_3) \equiv \varphi(m_1 \to cr_1, \varphi(R_2, R_3))$$

De ce fait, nous pouvons en déduire que :

$$\begin{aligned}
\varphi(\varphi(R_1,R_2),R_3) &\equiv \varphi(m \to \text{waiting}(\varphi(m_1 \to cr_1, R_2), mb), R_3)\\
&\equiv m \to \text{waiting}(\varphi(\varphi(m_1 \to cr_1, R_2), R_3), mb)\\
\text{et } \varphi(R_1,\varphi(R_2,R_3)) &\equiv m \to \text{waiting}(\varphi(m_1 \to cr_1, \varphi(R_2,R_3)), mb)\\
&\equiv m \to \text{waiting}(\varphi(\varphi(m_1 \to cr_1, R_2), R_3), mb)\\
&\equiv \varphi(\varphi(R_1,R_2),R_3)
\end{aligned}$$

Les preuves de la propriété d'associativité de la fonction de fusion comportementale concernant les autres règles de sémantique naturelle de la règle de fusion R6 sont en tout point identique aux preuves de l'associativité de la fonction de fusion comportementale présentées pour la règle de fusion R5. Nous ne les détaillons donc pas.

Ainsi la règle de fusion comportementale R6 est associative.

◇

Preuve de l'associativité de la règle de fusion d'un comportement réactif concurrentiel (R7) :
La fusion comportementale d'un comportement réactif concurrentiel avec un autre comportement réactif est décrite par la règle de sémantique naturelle suivante :

$$\frac{(m_1 \to cr_2, m_2 \to cr_3) : m \to cr_{2.3}}{(m_1 \to \text{concurrency}(cr_1, cr_2), m_2 \to cr_3) : m \to \text{concurrency}(s(cr_1), cr_{2.3}) \mid m_1 \in cr_2}$$

Soit $R_1 : m_1 \to \text{concurrency}(cr_1, cr_2)$ et $(R_2, R_3) \in \mathcal{G}^2_{-E}$.

Grâce à l'hypothèse de récurrence, nous avons :

$$\varphi(\varphi(m_1 \to cr_2, R_2), R_3) \equiv \varphi(m_1 \to cr_2, \varphi(R_2, R_3))$$

De ce fait, nous pouvons en déduire que :

$$\begin{aligned}\varphi(\varphi(R_1,R_2),R_3) &\equiv \varphi(m \to \text{concurrency}(s(cr_1),\varphi(m_1 \to cr_2,R_2)),R_3)\\ &\equiv m \to \text{concurrency}(s(cr_1),\varphi(\varphi(m_1 \to cr_2,R_2),R_3))\end{aligned}$$

$$\begin{aligned}\text{et } \varphi(R_1,\varphi(R_2,R_3)) &\equiv \varphi(m_1 \to \text{concurrency}(cr_1,cr_2),\varphi(R_2,R_3))\\ &\equiv m \to \text{concurrency}(s(cr_1),\varphi(m_1 \to cr_2,\varphi(R_2,R_3)))\\ &\equiv m \to \text{concurrency}(s(cr_1),\varphi(\varphi(m_1 \to cr_2,R_2),R_3))\\ &\equiv \varphi(\varphi(R_1,R_2),R_3)\end{aligned}$$

Ainsi la règle de fusion comportementale R7 est associative.

◇

Preuve de l'associativité de la règle de fusion d'un comportement réactif séquentiel (R8) :
La fusion comportementale de deux comportements réactifs séquentiels est définie par la fusion comportementale de l'un des deux comportements réactifs séquentiels avec la transformation de l'autre comportement réactif séquentiel en un comportement réactif concurrentiel.

Or la fusion comportementale d'un comportement réactif concurrentiel avec un autre comportement réactif est associative (preuve ci-dessus). Ainsi la règle de fusion comportementale R8 est associative.

◇

Preuve de l'associativité de la règle de fusion d'un comportement réactif conditionnel (R9) :
La fusion comportementale d'un comportement réactif conditionnel avec un autre comportement réactif est notamment décrite par la règle de sémantique naturelle suivante :

$$\frac{\begin{array}{l}(m_1 \to cr_1, m_2 \to cr_3) : m \to cr_{1.3}\\ (m_1 \to cr_2, m_2 \to cr_3) : m \to cr_{2.3}\end{array}}{(m_1 \to \text{ifThenElse}(c,cr_1,cr_2), m_2 \to cr_3) : m \to \text{ifThenElse}(s(c),cr_{1.3},cr_{2.3})}$$

Soit $R_1 : m_1 \to \text{ifThenElse}(c,cr_1,cr_2)$ et $(R_2,R_3) \in \mathcal{G}^2_{\neg E}$.

Grâce à l'hypothèse de récurrence, nous avons :

$$\begin{aligned}\varphi(\varphi(m_1 \to cr_1,R_2),R_3) &\equiv \varphi(m_1 \to cr_1,\varphi(R_2,R_3))\\ \varphi(\varphi(m_1 \to cr_2,R_2),R_3) &\equiv \varphi(m_1 \to cr_2,\varphi(R_2,R_3))\end{aligned}$$

De ce fait, nous pouvons en déduire que :

$$\begin{aligned}\varphi(\varphi(R_1,R_2),R_3) &\equiv \varphi(m \to \text{ifThenElse}(s(c),\varphi(m_1 \to cr_1,R_2),\varphi(m_1 \to cr_2,R_2)),R_3)\\ &\equiv m \to \text{ifThenElse}(s(c),\varphi(\varphi(m_1 \to cr_1,R_2),R_3),\varphi(\varphi(m_1 \to cr_2,R_2),R_3))\end{aligned}$$

$$\begin{aligned}\text{et } \varphi(R_1,\varphi(R_2,R_3)) &\equiv \varphi(m_1 \to \text{ifThenElse}(c,cr_1,cr_2),\varphi(R_2,R_3))\\ &\equiv m \to \text{ifThenElse}(s(c),\varphi(m_1 \to cr_1,\varphi(R_2,R_3)),\varphi(m_1 \to cr_2,\varphi(R_2,R_3)))\\ &\equiv m \to \text{ifThenElse}(s(c),\varphi(\varphi(m_1 \to cr_1,R_2),R_3),\varphi(\varphi(m_1 \to cr_2,R_2),R_3))\\ &\equiv \varphi(\varphi(R_1,R_2),R_3)\end{aligned}$$

La preuve de la propriété d'associativité concernant la seconde règle de fusion comportementale d'un comportement réactif conditionnel avec un autre comportement réactif est en tout point identique et n'est donc pas présentée.

Ainsi la règle de fusion comportementale R9 est associative.

◇

Preuve de l'associativité de la règle de fusion d'un comportement réactif de traitement des exceptions (R10) :
La fusion comportementale de deux comportements réactifs de traitements de la même exception est décrite par la règle de sémantique naturelle suivante :

$$\frac{\begin{array}{l}(m_1 \to cr_1, m_2 \to cr_3) : m \to cr_{1.3}\\ (m_1 \to cr_2, m_2 \to cr_4) : m \to cr_{2.4}\end{array}}{(m_1 \to \text{try}(cr_1,e,cr_2), m_2 \to \text{try}(cr_3,e,cr_4)) : m \to \text{try}(cr_{1.3},e,cr_{2.4})}$$

Soit $R_1 : m_1 \to \text{try}(cr_1,e,cr_2)$, $R_2 : m_2 \to \text{try}(cr_3,e,cr_4)$, et $R_3 : m_3 \to \text{try}(cr_5,e,cr_6)$.

Grâce à l'hypothèse de récurrence, nous avons :

$$\begin{aligned}&\varphi(m_1 \to cr_1,\varphi(m_2 \to cr_3,m_3 \to cr_5)) \equiv \varphi(\varphi(m_1 \to cr_1,m_2 \to cr_3),m_3 \to cr_5)\\ &\varphi(m_1 \to cr_2,\varphi(m_2 \to cr_4,m_3 \to cr_6)) = cr_{(2.4).6} \equiv \varphi(\varphi(m_1 \to cr_2,m_2 \to cr_4),m_3 \to cr_6) = cr_{2.(4.6)}\end{aligned}$$

De ce fait nous pouvons en déduire que :

$$\begin{aligned}\varphi(\varphi(R_1,R_2),R_3) &\equiv \varphi(m \to \text{try}(\varphi(m_1 \to cr_1, m_2 \to cr_3), e, cr_{2.4}), R_3)\\ &\equiv m \to \text{try}\left(\varphi(\varphi(m_1 \to cr_1, m_2 \to cr_3), m_3 \to cr_5), e, cr_{(2.4).6}\right)\end{aligned}$$

$$\begin{aligned}\text{et } \varphi(R_1,\varphi(R_2,R_3)) &\equiv \varphi(R_1, m \to \text{try}(\varphi(m_2 \to cr_3, m_3 \to cr_5), e, cr_{4.6}))\\ &\equiv m \to \text{try}\left(\varphi(m_1 \to cr_1, \varphi(m_2 \to cr_3, m_3 \to cr_5)), e, cr_{2.(4.6)}\right)\\ &\equiv m \to \text{try}\left(\varphi(\varphi(m_1 \to cr_1, m_2 \to cr_3), m_3 \to cr_5), e, cr_{(2.4).6}\right)\\ &\equiv \varphi(\varphi(R_1,R_2),R_3)\end{aligned}$$

Les preuves de la propriété d'associativité de la fonction de fusion comportementale concernant les autres règles de sémantique naturelle de la règle de fusion R10 sont en tout point similaire à la preuve de l'associativité de la fonction de fusion comportementales présentées ci-dessus. Nous ne les détaillons donc pas.

Ainsi la règle de fusion comportementale R10 est associative.

◇

Nous venons par conséquent de vérifier que toutes les règles de fusion comportementale vérifiaient la propriété d'associativité :

$$\forall (r_1,r_2,r_3) \in \mathcal{R}^3 \quad \varphi(\varphi(r_1,r_2),r_3) \equiv \varphi(r_1,\varphi(r_2,r_3))$$

□

6.7 Conclusion

Dans ce chapitre, nous avons présenté :

- Le mécanisme de combinaison des comportements réactifs, la fusion comportementale, permettant de générer un comportement réactif exécutable disposant de la même sémantique d'exécution que l'ensemble des comportements réactifs combinés. Ce mécanisme est utilisée lors de l'héritage des schémas d'interactions et lors de l'exécution des comportements réactifs associés à un message déclencheur.
- Les deux propriétés essentielles de la fusion comportementale lui conférant son fort pouvoir d'expression de la sémantique des comportements réactifs : la commutativité et l'associativité.

Apports de notre modèle à interactions distribuées

Les interactions disposent d'un niveau d'abstraction et de conception similaire à celui des objets grâce à la notion de schéma d'interactions. Ainsi les interactions sont des entités de même statut et importance que les objets. Elles sont *indépendantes* et *autonomes* vis-à-vis des objets interagissants sur lesquels elle s'appliquent. Ceci permet notamment l'expression des interactions au niveau des instances sans pour autant impliquer que toutes les instances d'une classe soient concernées.

De plus, cette séparation logique entre les objets et les interactions permet de préserver le principe d'encapsulation, les comportements réactifs étant exclusivement exprimés sur l'interface des objets participants. Cette séparation permet de fortement réduire la complexité des objets interagissants et améliore la réutilisation et la maintenance de ces objets. La mise en œuvre des interactions sous la forme d'objets (étendus) permet de les doter d'informations propres (aussi bien au niveau structurel que fonctionnel) et de définir une interaction sur une autre interaction.

Les interactions sont n-aires et non orientées : chaque participant pouvant aussi bien être l'instigateur du déclenchement d'un comportement réactif (par le biais de l'une de ses méthodes) que l'un des acteurs de l'exécution d'un comportement réactif. De plus, afin d'améliorer l'expressivité des comportements réactifs, ces derniers sont réifiés sous la forme de règles d'interactions. L'interprétation de ces règles est spécifiée par des opérateurs. Chaque opérateur, également réifié, définit le sens des actions et du contrôle des messages. Ainsi l'expression des comportements réactifs est adaptable, par la modification de la sémantique des opérateurs, et extensible, par l'ajout de nouveaux opérateurs [6].

6. Ceci implique bien évidemment de vérifier que la sémantique définie pour ces nouveaux opérateurs conserve toujours les propriétés de commutativité et d'associativité de la fonction de fusion comportementale.

Mise en œuvre du modèle

Notre modèle à interactions distribuées est basé d'une part sur le contrôle de l'envoi de messages (puisque sa sémantique est modifiée par la prise en compte des comportements réactifs) et d'autre part sur une approche dynamique du support des interactions, et notamment sur un mécanisme de découverte dynamique des schémas d'interactions.

Le modèle présenté décrit de manière idéale le support des interactions dans un environnement compilé et fortement typé. Cependant, certains aspects, comme la gestion des exceptions, sont très dépendants du langage utilisé ou difficilement réalisables. Étant conscient de ces limitations imposées par la mise en œuvre, elles ont été prises en compte afin que les concepts primordiaux du modèle restent valides dans toute mise en œuvre. Nous pensons donc que le modèle à interactions distribuées présenté peut être mis en œuvre dans différents langages à objets compilés et typés.

La partie suivante complète la description de l'architecture du modèle à interactions distribuées en vue de sa mise en œuvre dans un environnement compilé, fortement typé et distribuée. Le chapitre 9 présente plusieurs mécanismes pour contrôler l'envoi de messages dans les langages n'offrant pas, à priori, cette possibilité (ces mécanismes sont issus de [Duc97a]), tandis que l'annexe A présente un service (au sens distribué du terme) de dépôt de schémas d'interactions.

Nous présentons, dans la prochaine partie, l'architecture globale de mise en œuvre du modèle à l'aide de C++ [Str91] et CORBA [OMG98].

Partie III

UNE ARCHITECTURE DE MISE EN ŒUVRE

Dans cette partie vous trouverez ...

Cette partie décrit une mise en œuvre de l'architecture et du protocole que nous avons proposé pour la gestion des interactions. Ainsi nous montrons comment les interactions s'intègrent dans le modèle à objets du langage C++ et dans l'architecture distribuée de CORBA.

Le **chapitre 7** présente les deux principales techniques utilisées par notre architecture pour mettre en œuvre les interactions. La première technique est la réflexivité. Elle est plus particulièrement présentée dans les langages à classes. Ses deux principales propriétés, l'*introspection* et l'*intercession*, sont définies. La seconde technique est la programmation par aspects – *Aspect Oriented Programming* (AOP) [KLM+97]. Son rôle est « d'extraire » du code des objets des propriétés orthogonales au comportement intrinsèque de ces objets.

Le **chapitre 8** présente la manière dont les interactions et les schémas d'interactions sont représentés dans notre architecture. De cette représentation, nous pouvons définir un ensemble de classes (et méta-classes) définissant les fondations de notre architecture (noyau minimal). De ces fondations, ce chapitre décrit le protocole, existant à l'exécution, permettant la gestion des interactions.

Le **chapitre 9** se concentre, quant à lui, sur la gestion de l'exécution des interactions dans le langage compilé et fortement typé qu'est C++ [Str91]. Il complète donc la description de l'architecture du modèle à interactions distribuées définie dans le chapitre précédent. Nous présentons dans ce chapitre notre solution pour contrôler les messages. Nous décrivons également notre solution pour évaluer de manière dynamique des messages dans un environnement compilé et fortement typé. Il se termine sur la description d'une mise en œuvre possible du dépôt de schémas d'interactions sous la forme d'un service CORBA en définissant un ensemble d'interfaces IDL (*Interface Definition Language* [Lam87]) permettant d'accéder aux méta-informations contenues dans un dépôt de schémas d'interactions.

REMARQUES. — Différentes mises en œuvres de ce modèle à interactions distribuées (ou de sous-parties du modèle à interactions distribuées) ont été réalisées ou sont en cours de réalisation (SmallTalk, Java RMI, C++ et CORBA, Java et CORBA). Nous présentons, dans cette partie, l'architecture globale de mise en œuvre du modèle. Le code utilisé pour illustrer notre discours est en C++ [Str91], Open C++ [Chi95], CorbaScript [MGG96] et CORBA [OMG98]. Cependant, le protocole proposé est adaptable à différents langages à objets et à différentes architectures distribuées. Ainsi, bien que la définition de l'architecture distribuée présentée soit décrite à l'aide du bus logiciel (*middleware*) CORBA, elle peut être aisément transposée dans un autre environnement distribué.

Chapitre 7

Contrôle de l'envoi de messages : apports des systèmes réflexifs

> « *Reflection : an entity's integral ability to represent, operate on, and otherwise deal with itself in the same way that it represents, operates on, and deals with its primary subject matter.* » [Smi90]
>
> « *The coordination languages have their focus on separation between coordination and computation, not meaning that they are independent or dual concepts, but meaning that they are orthogonal.* » [CR97]

Il n'est plus nécessaire de présenter les propriétés qui caractérisent les langages à objets, ainsi que leurs domaines d'applications. Cependant le terme *réflexion* mérite que l'on s'y attarde. Ce concept, bien que possédant différentes connotations en fonction du domaine d'application, exprime, d'une manière générale, *la capacité d'un système à raisonner et à agir sur lui-même*. Ainsi, ce chapitre va présenter la réflexion dans les langages à objets et plus particulièrement dans les langages à classes. Les deux principales propriétés des langages réflexifs, l'*introspection* et l'*intercession*, seront définies, puis une classification des techniques de réflexion sera proposée.

La programmation par aspects – *Aspect Oriented Programming* [KLM+97] (AOP) – est ensuite présentée. Son rôle est de permettre « d'extraire » du code des objets des propriétés orthogonales au comportement intrinsèque des objets. Ces propriétés sont généralement des propriétés qui affectent, de manière implicite, les performances ou la sémantique du système et peuvent rarement être considérées comme des unités de la décomposition fonctionnelle du système. Elles sont nommées *aspects*. Ce chapitre conclut par une comparaison de la prise en compte de ces propriétés orthogonales et de leur mise en œuvre entre la programmation par aspects et la programmation par objets classique.

7.1 Contrôle de l'envoi de messages

Dans la définition du modèle à interactions distribuées, présentée au chapitre 5, l'unique moyen offert aux objets pour communiquer entre eux, le concept d'envoi de messages du modèle à objets, est étendu (définition 5.1) pour prendre en compte les interactions. En effet, la présence d'une interaction implique l'exécution de comportements réactifs en lieu et place des comportements définis par les méthodes qui leur sont associées, la sémantique traditionnelle étant conservée en l'absence d'interaction.

Modifier la sémantique de l'envoi de messages pour offrir un support aux interactions implique la mise en place d'un ensemble de structures et de comportements spécifiques pour contrôler l'envoi de messages. De plus, la granularité du contrôle doit être suffisamment fine pour permettre de contrôler

individuellement chaque méthode de chaque objet du système. Cependant, ce contrôle est dépendant de chaque langage à objets visé. De plus, ce contrôle implique la mise en place d'un mécanisme particulier de capture des messages. Si, dans certains langages (notamment les langages dits ouverts), cette mise en place peut être réalisée sans aucune intervention du programmeur, dans d'autres langages (par exemple C++ [Str91] ou Java [CH96]) elle doit être, a priori, explicitement réalisée par le programmeur.

Ainsi, un objet devient interagissant lorsque la sémantique de l'envoi de messages à destination de cet objet supporte la gestion de l'exécution des comportements réactifs définis par le modèle à interactions distribuées.

Or, la modification de la sémantique d'un des aspects du langage implique l'utilisation d'un langage extensible permettant aux programmeurs d'en spécialiser les aspects fondamentaux (pour en modifier la sémantique) et, dans notre cas, plus particulièrement l'envoi de messages. Cette modification peut être réalisée de manière transparente par le programmeur pour les langages ouverts, mais est généralement impossible pour les autres langages (par exemple en C++, il est impossible, a priori, de modifier la sémantique de l'envoi de messages). Pour ces langages, il est nécessaire d'avoir recours à des outils permettant « d'ouvrir » ces langages et donnant accès à ces mécanismes internes. L'utilisation d'outils définissant un protocole à métaobjet (MOP), tel qu'Open C++ [Chi95] pour le langage C++, est une solution à cette « ouverture ».

7.2 Réflexivité dans les langages à objets

En informatique, les domaines de l'intelligence artificielle et des langages de programmation sont les premiers à avoir introduit la notion de réflexion afin d'offrir un environnement flexible et adaptatif permettant la modification de la sémantique du langage lui-même. En programmation, bien que la réflexion ne se limite pas exclusivement aux langages à objets, puisqu'elle existe aussi en programmation logique ou fonctionnelle [DM95], elle a trouvé un terrain très favorable dans la technologie objet.

Ce concept de réflexion fut introduit pour la première fois dans un langage de programmation par B. SMITH avec son langage 3-Lisp [Smi84], une extension réflexive du langage Lisp [Ste90]. B. SMITH définit dans ce langage les concepts de tours et de procédures réflexives afin que les programmes écrits en 3-Lisp puissent modifier leur propre exécution. Dans l'absolu, une tour réflexive se compose d'une infinité d'interprètes. Chacun d'eux exécute l'interprète du niveau juste inférieur tandis que l'interprète de base exécute le programme de l'utilisateur [WF86, MJD96]. Les procédures réflexives permettent de passer d'un niveau à un autre en effectuant des calculs réflexifs.

Notion de métaobjet

En 1987, dans son mémoire de thèse, P. MAES a proposé des définitions précises caractérisant la programmation réflexive. La formalisation de ces nombreuses définitions a donné lieu à la réalisation du langage 3-KRS [Mae87a] introduisant dans un langage de représentation des connaissances une architecture réflexive à la 3-Lisp. Dans le modèle de 3-KRS, où tout objet est associé à un *métaobjet* [1], il existe un lien bijectif entre le métaobjet et son objet.

De son modèle [Mae87b], P. MAES a déduit le fait qu'il faut distinguer le niveau objet (ou niveau de base) du niveau réflexif (ou niveau méta) : le niveau objet manipule les données et les opérations concernant le domaine de l'application tandis que le niveau réflexif manipule des informations concernant la mise en œuvre et l'exécution de l'application.

Du concept de métaobjet, les auteurs de [BKdR91] ont déduit un protocole permettant une manipulation des métaobjets, il s'agit du *protocole à métaobjets – Meta Object Protocol* (MOP). Le MOP est une seconde interface du langage qui supprime la distinction entre les utilisateurs du langage et ses concepteurs. En effet, il offre aux utilisateurs la possibilité de modifier, de façon incrémentale, le comportement du langage ainsi que sa mise en œuvre (figure 7.1). Un MOP peut notamment être défini par un ensemble de métaobjets et représente certains aspects de la mise en œuvre (par exemple l'envoi de messages, la gestion de la concurrence, etc.) du langage pour lequel il est défini.

NOTE. — Un langage permettant au programmeur de spécialiser les métaobjets fournis en standard pour définir de nouvelles sémantiques de représentation (sémantique structurelle) et d'exécution (sémantique comportementale) est dit *ouvert*.

1. « *Les éléments primitifs d'un langage de programmation – les classes, les méthodes et les fonctions génériques – sont rendus manipulables en tant qu'objets. Parce que ces objets représentent des portions de programme, on les nomme par* ***métaobjets****.* » [BKdR91]

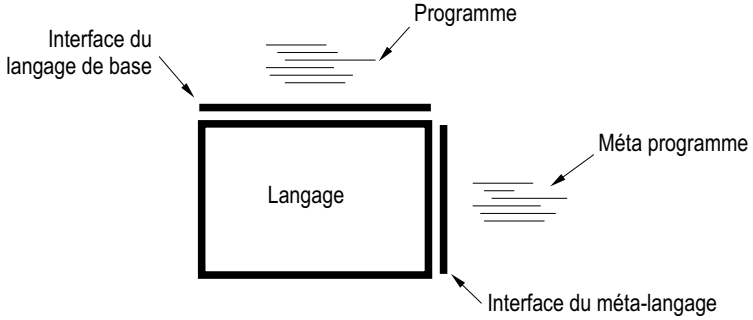


FIG. 7.1 – *Mise en œuvre dite « ouverte »*

Introspection et intercession

Lorsque l'on parle de programmation réflexive, on distingue traditionnellement deux aspects pour notamment spécialiser les métaobjets fournis en standard [SF96] : celui permettant de définir de nouvelles sémantiques de représentation (réflexion *structurelle*) et celui permettant de modifier la sémantique de l'exécution (réflexion *comportementale*).

La réflexion structurelle permet d'obtenir une représentation du langage (en vue de *l'introspection*) en proposant un mécanisme pour encoder (et manipuler) les données propres à l'exécution, la *réification* [Coi87]. La réflexion comportementale permet, quant à elle, d'agir sur les mécanismes de base du langage (permettant ainsi *l'intercession*) [MC93].

7.2.1 Introspection : réflexion structurelle

> « *L'introspection est la possibilité qu'a un programme d'observer et donc de raisonner sur son propre état.* » [BGW93]

La réflexion structurelle offre à un système réflexif un moyen permettant son *introspection* et son auto représentation. On utilise le langage réflexif lui-même pour gérer le contrôle des programmes [BG96] grâce à la réification des concepts du langage qui peuvent ainsi être introspectés. De ce fait, le langage réflexif est dit *homogène* puisque l'écriture et le contrôle des programmes sont réalisés à l'aide d'un seul et même langage. Ainsi, les différentes caractéristiques de représentation (statique) et d'exécution (dynamique) des programmes sont rendues concrètes dans le langage pour être manipulées et inspectées.

Le modèle Smalltalk-76 [Ing78] a été l'un des premiers à introduire la notion de réflexion structurelle. Il définit un langage homogène où toute entité manipulée est un objet. De ce fait, chaque classe est considérée comme un objet à part entière et est une instance d'une autre classe, nommée *métaclasse*. Cette dernière définie la structure et le comportement de la classe (tel que, par exemple, la sémantique de l'instanciation). Par la suite, le modèle ObjVLisp [Coi87] a unifié le concept de classes et métaclasses. De plus, il offre un mécanisme permettant de disposer d'un nombre non défini et non limité de méta-niveaux.

Certains modèles mettent en œuvre le paradigme du « tout objet ». C'est le cas de Smalltalk ou CLOS [BDG+88]. Ainsi les programmes disposent d'une représentation d'eux-mêmes à l'exécution sous la forme d'un ensemble d'objets. Par exemple, les méthodes en Smalltalk, ou les fonctions génériques en CLOS, sont des instances d'une classe qui représente le code de la méthode ou de la fonction générique.

Smalltalk va même plus loin au niveau de la réflexion structurelle puisqu'il réifie entièrement la sémantique de son code, de son noyau, de son environnement (y compris les outils de développement tels que les butineurs de code) et de son mode d'exécution. Ceci fait de Smalltalk l'un des langages, si ce n'est le langage, le plus réflexif structurellement [Led98]. D'ailleurs, récemment, les « pères » de Smalltalk, A. KAY et D. INGALLS, ont définis Squeak [IKM+97], un avatar de Smalltalk permettant de réifier la machine virtuelle de Smalltalk en Smalltalk lui-même.

Bien que les langages réflexifs soient principalement ceux qui ont été influencés par la communauté Lisp, certains langages à objets issus du courant de l'école Scandinave (Java, etc.) offrent un support à la réflexivité structurelle. On peut citer Java [CH96], où les classes et les interfaces sont des objets de première classe. Cette propriété permet d'introspecter les classes et les objets manipulés par la machine virtuelle. De plus, grâce aux API *Reflection* [Fla97], il est possible d'introspecter la représentation des objets du langage de manière plus complète (méthodes, etc).

Le concept d'introspection des langages réflexifs permet d'inspecter la représentation du langage (par exemple connaître la classe d'un objet ou la liste de ses méthodes) mais ne permet pas de modifier cette représentation.

7.2.2 Intercession : réflexion comportementale

> « *L'intercession est la possibilité qu'a un programme de modifier sa propre exécution, ou de modifier son interprétation.* » [BGW93]

Les protocoles à métaobjets ont été initialement définis comme des interfaces supplémentaires des langages de programmation pour donner à l'utilisateur la possibilité de modifier le comportement et la mise en œuvre du langage. Nous appelons ce concept la réflexion comportementale, ou *intercession* [BGW93].

Ainsi, grâce à un langage supportant la réflexion comportementale, de nouvelles sémantiques non prévues lors de la conception du langage et correspondantes à des besoins spécifiques peuvent être introduites par modification, adaptation ou extension du langage. Ceci permet, par exemple, de définir une nouvelle sémantique d'envoi de messages lorsque les objets sont distants [CM93] ou de modifier le modèle initial (c'est le cas d'Actalk [Bri99] qui définit un modèle d'acteurs en Smalltalk [Dug90]).

L'aspect dynamique de certains langages réflexifs permet d'intervenir, à l'exécution, sur les caractéristiques du langage. Ceci permet ainsi de modifier, au cours de l'exécution d'une application, l'un des comportements du langage. Par exemple, dans un système en cours de tests et d'optimisations, on cherche à obtenir une trace de tous les envois de messages à la suite de la découverte d'une bogue pour le localiser en « remplaçant » le métaobjet d'envoi de messages par un métaobjet d'envoi de messages tracé. Ces langages réflexifs sont ainsi dotés d'une capacité leur permettant de s'adapter dynamiquement à une nouvelle situation.

L'une des propriétés de la réflexion comportementale est qu'elle permet d'agir sur les mécanismes de base du langage en offrant un accès à l'interprète du langage et aux données utilisées par ce dernier lors de l'exécution d'un programme. Les auteurs de [LKR+92] proposent une approche basée sur cette propriété qui consiste à fournir un processeur (interprète ou compilateur) « ouvert » constitué d'un ensemble de petites unités indépendantes entre elles et entièrement modifiables.

Cette approche permet de modifier la sémantique du langage en redéfinissant une, ou plusieurs, de ces unités. Ceci permet, entre autre, d'étendre le langage ou d'optimiser le code généré (l'article [Chi97] montre un exemple d'optimisation d'un code réalisant un calcul matriciel). De ce fait, il est possible, grâce aux métaobjets du langage, de modifier le comportement des objets du programme en contrôlant la compilation (ou l'interprétation) du programme.

EXEMPLE. — Open C++ [Chi95], une extension réflexive de C++ lors de la compilation, permet de réifier l'arbre de syntaxe abstraite d'un programme. Ainsi, le programmeur a la possibilité d'introspecter le code de l'application. Cependant cette introspection est réalisée sur les classes et non sur les objets eux-mêmes, puisque le MOP d'Open C++ n'est disponible que lors de la phase de compilation.
Open C++ propose également, sous la forme de métaclasses, la construction des nœuds de l'arbre syntaxique d'un programme C++ et permet de modifier ce dernier (à travers son arbre syntaxique). Ainsi, la sémantique du langage C++ est modifiable à travers ce pré-compilateur. Il est donc possible d'étendre la syntaxe du langage C++ ou d'introduire de nouvelles sémantiques (par exemple l'envoi de messages à distance) par simple extension du pré-compilateur.

La réflexion comportementale a pris un essor tardif comparé à la réflexion structurelle. Ceci est la conséquence principale de la difficulté de mettre en œuvre de manière efficace la complexité intrinsèque de la réflexion comportementale. Cependant, grâce à la réflexion comportementale, il est possible, par réification complète du processeur (interpréteur ou compilateur), d'avoir un contrôle complet sur la sémantique du langage.

7.2.3 Différents mécanismes de contrôle de l'envoi de messages

Il existe une multitude de façons de réaliser le contrôle de l'envoi de messages [Mae87a, Fer89, BKdR91, CM93, McA95]. Dans ce paragraphe, nous présentons les trois mécanismes permettant un contrôle de l'envoi de messages les plus souvent utilisés et les plus représentatifs.

Redéfinition de méthodes par sous-classage

Une solution souvent mise en œuvre pour contrôler les messages consiste à sous-classer la classe des objets à contrôler. Ce sous-classement permet de redéfinir les méthodes de la classe et ainsi rendre possible l'introduction d'un mécanisme de contrôle des messages.

L'un des inconvénient de cette approche est la définition d'un nombre non négligeable de classes dont le rôle se cantonne à prendre le contrôle des messages. En effet, ces classes générées et les méthodes redéfinies sont, bien souvent, vides d'une réelle sémantique.

Le contrôle de seulement certaines instances d'une classe implique, lorsque le langage le permet [Riv95, Riv96, Ser99], un changement de classe des instances contrôlées. Lorsqu'un tel changement n'est pas possible alors le sous-classement doit être réalisé soit par un transtypage, soit lors de la phase de conception. Dans ce dernier cas, le contrôle s'applique sur toutes les instances de la classe.

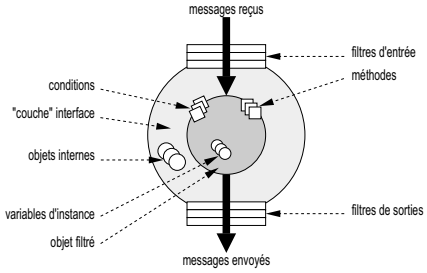
Le sous-classage permet également de contrôler les méthodes qui n'appartiennent pas à l'interface de l'objet contrôlé et outrepasse, dans une certaine mesure (grâce aux méthodes `protected` notamment), le principe d'encapsulation de l'objet. Il n'offre, de plus, aucune abstraction du contrôle, ce dernier étant fortement lié à l'objet contrôlé puisque défini dans sa classe (ou, plus exactement l'une de ses sous-classes).

EXEMPLE. — La mise en œuvre de FLO [Duc97b] dans NEOCLASSTALK [Riv95] est basée sur cette solution. En effet, elle utilise les possibilités de changement dynamique de classe et de spécialisation du compilateur Smalltalk afin de modifier la sémantique des méthodes. Ainsi lorsqu'une instance doit être contrôlée, elle change de classe, sa nouvelle classe étant une sous-classe de la classe initiale permettant le contrôle des messages. Il est à noter que ce sous-classage est réalisé automatiquement par le système et non à la charge du programmeur.

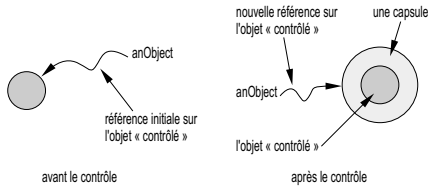
Encapsulation et filtrage

Une autre solution consiste à encapsuler les objets devant être contrôlés. Cet encapsulation est basée sur une substitution physique de l'identité de l'objet contrôlé par une capsule [Pas86], ou un filtre [ABV92]. Les messages envoyés à l'objet contrôlé sont en fait envoyés à la capsule ou au mécanisme de filtrage.

Les approches basées sur l'utilisation d'une capsule (figure 7.2b ci-dessous) définissent celle-ci comme étant un objet minimal [Bri89, PWG93] ne connaissant aucun message. Ainsi, le mécanisme de traitement des erreurs d'envoi de messages est invoqué (en Smalltalk, par exemple, il s'agit de la méthode `doesNotUnderstand:`) et permet ainsi de contrôler les messages.



7.2a – Contrôle des messages par le biais du langage (notion de filtres)



7.2b – Contrôle des messages par le biais d'un objet « capsule »

FIG. 7.2 – *Contrôle des messages par des capsules ou des filtres*

Dans le cas des approches utilisant des filtres, les messages envoyés à un objet contrôlé sont interceptés et filtrés (figure 7.2a ci-dessus). Le contrôle des messages est réalisée lors du filtrage. Par exemple, dans le modèle des filtres de composition [AW+93], chaque filtre spécifie comment un message reçu par le mécanisme de filtrage doit être transmis au filtre suivant ou à l'objet destinataire du message.

Cette solution implique, notamment dans le cas des approches avec capsule, un changement dynamique des références des objets contrôlés. De plus, le sur-coût apporté par le contrôle est non négligeable (puisqu'il implique une double recherche du message, une première fois dans l'objet capsule ou à travers les filtres, et une seconde fois dans l'objet encapsulé).

Ces notions de filtrage ou d'objet capsule, ne permettent pas, en raison du concept d'encapsulation du modèle à objets, de contrôler les messages n'appartenant pas à l'interface des objets contrôlés. En effet, le contrôle est défini hors de l'objet contrôlé.

Ceci implique, et ce n'est pas anodin, que seuls les messages émis depuis l'extérieur de l'objet peuvent être contrôlés [PWG93]. Enfin, ces approches n'offrent aucune abstraction du contrôle en ne fournissant pas de véritable distinction entre l'objet contrôlé et le contrôle lui-même (puisque les deux sont très fortement liés).

Utilisation de la réflexivité et des métaobjets

D'autres approches permettent un contrôle des messages en se basant sur les concepts définis par P. MAES dans [Mae87b, Mae88]. Parmi ces concepts, P. MAES introduit le fait qu'un objet est associé à un métaobjet qui contient toutes les informations réflexives, dont notamment la manière dont l'objet réagit aux messages. Ainsi, il est possible d'associer aux objets, dont certains messages doivent être contrôlés un métaobjet spécifique qui définit ce contrôle.

Dans les langages à objets, le mécanisme d'envoi de messages est l'unique moyen offert aux objets pour communiquer entre eux. Ainsi, le contrôle de l'envoi de messages est donc l'une des préoccupations majeures de tous les systèmes réflexifs offrant un support à la réflexion comportementale [MC93]. Or, l'exécution d'un message fait partie intégrante du mécanisme d'envoi de messages [McA95].

EXEMPLE.— La première version d'Open C++ [Chi93] propose une abstraction de l'envoi de messages en associant un métaobjet aux objets dont le comportement doit être contrôlé. Lors de l'invocation d'une méthode réflexive sur l'objet contrôlé la méthode `Meta_MethodCall` du métaobjet est exécutée. C'est en redéfinissant cette méthode que le programmeur peut contrôler les messages.

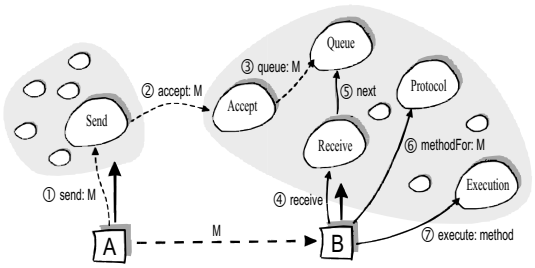


FIG. 7.3 – *Contrôle des messages par le biais de métaobjets (ici ceux de* CODA*)*

Il est à noter que certaines approches associent plusieurs métaobjets pour le contrôle des messages (figure 7.3 ci-contre), chaque métaobjet réalisant une étape du mécanisme d'envoi de messages. De plus, certaines approches, comme par exemple le langage à prototypes MOOSTRAP [MC93], associe à chaque objet (y compris les métaobjets) un métaobjet. Par conséquent, le mécanisme de contrôle de l'envoi de messages définit par ces approches peut s'avérer très coûteux aussi bien en temps d'exécution qu'en place mémoire.

EXEMPLE.— Le MOP de CODA [McA95] est un MOP spécialisé dans l'abstraction de la communication entre objets. Il propose en effet sept méta-composants (figure 7.3) permettant de contrôler très finement le mécanisme d'envoi de messages (de l'envoi proprement dit jusqu'à l'exécution du message par l'objet destinataire).

Support du contrôle de l'envoi de messages dans les langages réflexifs

Parmi les premiers langages à avoir introduit le contrôle de l'envoi de messages, on peut citer Smalltalk. À l'aide de la méthode `doesNotUnderstand:` [GR83], il permet de définir la sémantique associée à la réception d'un message non connu. Ainsi, en substituant un objet *minimal* sans méthodes (provoquant par conséquent un appel à la méthode `doesNotUnderstand:` à chaque invocation), il est possible de définir de nouvelles sémantiques à l'envoi de messages (par exemple l'invocation distante [McC87]). L'article [Duc97a] décrit d'autres techniques de contrôle de l'envoi de messages en Smalltalk.

Quant à lui, le langage 3-KRS [Mae87a] fournit à chaque objet un métaobjet gérant l'intégralité des concepts du modèle (héritage, envoi de messages, etc.). Ainsi, un message reçu par un objet est automatiquement transmis par ce dernier à son métaobjet. De ce fait, la modification des métaobjets permet de définir de nouvelles sémantiques non envisagées à l'origine.

Une technique similaire est proposée par la première version d'Open C++ [CM93, Chi93] qui offre un métaobjet de gestion des envois de messages. Celui-ci réifie l'appel d'une méthode sur un objet et permet d'appliquer un traitement spécifique sur cet appel (par exemple modifier les paramètres, etc.). La version 2 d'Open C++ [Chi95] permet, quant à elle, de modifier l'arbre de syntaxe abstraite d'un programme C++. Ainsi, elle permet également de modifier le mécanisme d'envoi de messages. Cependant, aucun métaobjet gérant les envois de messages n'est disponible [2]. L'extension de l'envoi de messages s'effectue donc par la modification du code C++ réalisant l'envoi de messages.

CODA [McA95] utilise, lui aussi, la notion de métaobjet pour permettre le contrôle de l'envoi de messages (figure 7.3). Cependant, ce contrôle n'est pas encapsulé par un unique métaobjet (comme cela est le cas avec les langages ci-dessus) mais par un ensemble de sept métaobjets. Ils disposent tous d'un rôle particulier dans un environnement concurrent et distribué. CODA offre donc une granularité très fine du contrôle de l'envoi de messages contre une complexité accrue de mise en œuvre.

2. En effet, la gestion de l'envoi de messages a lieu durant l'exécution du programme, or le MOP d'Open C++ v2 s'applique durant la phase de compilation.

En se concentrant principalement sur la communication entre objets, CODA propose un protocole dont le pouvoir d'expression est beaucoup plus riche que la plupart des MOP. Il permet de distinguer clairement chacune des étapes de la communication entre objets. Ainsi, il entre en jeu dès l'émission des messages, et pas seulement à leur réception comme cela est le cas dans les autres approches. De plus, le MOP de CODA n'est lié à aucun langage à objets spécifique et peut être perçu comme un modèle de MOP pour le contrôle de l'envoi de messages.

7.2.4 Capture de l'envoi de messages : l'approche d'Open C++

Nous venons de présenter, dans le paragraphe précédent, différents mécanismes de contrôle des messages. Nous présentons dans ce paragraphe la solution proposée par la première version d'Open C++ [Chi93]. Elle est en effet très proche de notre propre solution.

Open C++, une extension du langage C++, permet d'associer un métaobjet à certains objets. Cette association est spécifiée dans la classe de l'objet (ce qui implique que toutes les instances seront contrôlées) et permet de contrôler les appels des méthodes et les accès aux variables (figure 7.4 ci-contre).

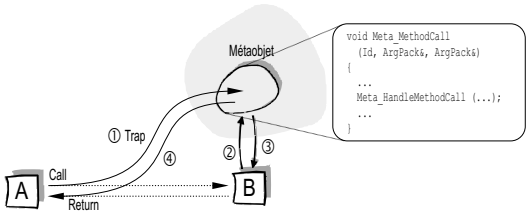


FIG. 7.4 – *Contrôle des messages avec* OPEN C++

La philosophie de P. MAES [Mae87b] stipule que le métaobjet est responsable de la phase de recherche des méthodes. L'approche d'Open C++ diffère singulièrement sur ce point puisque la phase de recherche des méthodes est laissée à la charge du langage (C++ en l'occurrence).

Ainsi la prise en compte du contrôle des messages consiste à rendre réflexives certaines méthodes. Dans le code C++, ceci est réalisé en précisant, sous la forme d'un commentaire particulier, que l'exécution de la méthode doit être contrôlée. L'exemple suivant montre comment mettre en œuvre cette prise de contrôle des messages (issu de l'exemple de [Chi93]) :

```
 1:  class Person
 2:  {
 3:    public:
 4:      Person (char *name, int age);
 5:      int Age ();
 6:  // MOP reflect :
 7:      int IncAge ();
 8:    private:
 9:      char *name;
10:      int   age;
11:  };
12:  // MOP reflect class Person : PrintMetaObject;
13:
14:  class PrintMetaObject ...
15:
16:  void PrintMetaObject::Meta_MethodCall (Id m_id, Id category, ArgPac& args, ArgPac& reply)
17:  {
18:    printf ("*** %s was called.\n", Meta_GetMethodName (m_id));
19:    Meta_HandleMethodCall (m_id, args, reply);
20:  };
```

Lorsqu'une méthode réflexive est invoquée (c'est le cas de la méthode de la ligne 7) alors le message est réifié et passé en paramètre de la méthode `Meta_MethodCall` du métaobjet[3] (le métaobjet utilisé est défini par la ligne 12). Les méthodes non réflexives étant traitées « normalement » (c'est-à-dire avec la sémantique d'envoi de messages du langage C++).

La méthode `Meta_MethodCall` du métaobjet `PrintMetaObject` est définie dans le code C++ (en effet le code du métaobjet est du code C++ classique). Elle prend comme paramètre une représentation réifiée du message à exécuter. Dans l'exemple ci-dessus, le contrôle du message consiste à afficher une trace de l'appel puis à exécuter la méthode contrôlée (grâce à la méthode `Meta_HandleMethodCall` du métaobjet).

3. L'accès au métaobjet est réalisée grâce à une primitive, nomméeTrap par la figure 7.4, et non à l'envoi de message, ceci afin d'éviter une réification récursive infinie.

NOTE. — Cette approche implique un contrôle de classes (*class-based control*) et non d'instances tout en offrant une granularité du contrôle au niveau des méthodes. Ainsi toutes les instances d'une classe sont, a priori, contrôlées. Cependant, pour des raisons de mise en œuvre, seules les instances créées dans le tas (par l'opérateur `new`) sont contrôlées. Il est donc possible, d'une certaine façon, de choisir si une instance doit être, ou non, contrôlée.

7.2.5 Liaisons réflexives

La liaison entre le niveau méta et le niveau de base peut être appliquée lors de chaque phase du développement d'une application (exécution, chargement, compilation, etc.). Pour toutes ces phases du développement, nous définissons une *liaison réflexive* spécifique et explicitons ses principales caractéristiques.

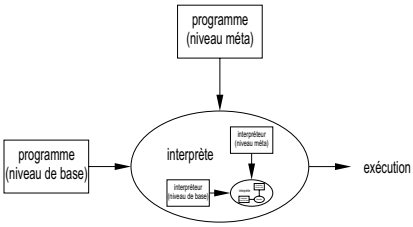


FIG. 7.5 – *Principe de la liaison réflexive à l'exécution (tour réflexive)*

Les premiers langages à introduire la réflexion sont des langages interprétés (Smalltalk notamment) où la liaison a lieu lors de la phase d'exécution (*liaison réflexive à l'exécution*). En effet, ces langages disposent d'une représentation du programme et de sa sémantique à l'exécution. Il leur suffit donc de permettre une adaptation de cette sémantique en fonction du programme au niveau méta.

Ainsi l'interprétation du programme au niveau de base est réalisée par un processeur sémantiquement modifié par le programme du niveau méta. Ce mécanisme permet de simuler le concept de la tour réflexive (figure 7.5). Cependant, pour des raisons évidentes de mise en œuvre, le nombre de niveaux est fini (mais pas nécessairement limité), le métaobjet du niveau le plus élevé n'étant pas réifié.

La liaison réflexive à l'exécution permet une adaptation dynamique du comportement d'interprétation du niveau de base par simple modification du programme du niveau méta. Elle permet aux langages réflexifs de s'adapter dynamiquement à une nouvelle situation. En contre partie de cette dynamicité, la liaison réflexive pêche par son manque d'efficacité dû principalement à l'interprétation réflexive des interpréteurs à chaque niveau de la tour réflexive. Il est à noter que de récents travaux [LKR+92, MMA+95, LK99] vont dans le sens de rendre les langages réflexifs *au moins aussi* efficaces que leurs équivalents non réflexif et donc de rendre caduc ce problème de manque d'efficacité de la réflexion à l'exécution tout en conservant la propriété d'adaptation dynamique propre à la liaison réflexive à l'exécution.

EXEMPLE. — Iguana [GC96] est un MOP, basé sur C++, existant à l'exécution (car, d'après ses concepteurs, les informations nécessaires à un système adaptable n'existent qu'à l'exécution). Pour ce faire, il propose toute un éventail de métaobjets disponibles à l'exécution. Cependant, afin de limiter le sur-coût d'exécution impliqué par la réification, il permet de moduler sa granularité en fonction des besoins des applications.

Pour supprimer ce problème d'efficacité, et pour les langages réflexifs compilés, il peut être intéressant de réaliser la liaison lors de la phase de compilation (*liaison réflexive à la compilation*). Avec cette solution, les métaobjets n'existent que durant la phase de compilation et sont absents durant l'exécution. De ce fait, l'efficacité de l'application lors de son exécution n'est pas modifiée, seule l'efficacité de la compilation est affectée (ainsi, on déplace le sur-coût lié à la réflexion à la phase de compilation).

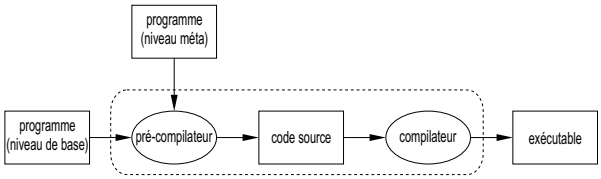


FIG. 7.6 – *Principe de la liaison réflexive à la compilation*

La figure 7.6 présente une architecture possible pour la liaison réflexive à la compilation (il s'agit de l'architecture utilisée par Open C++ ou Open Java). Une autre architecture possible consiste à définir un compilateur « ouvert » produisant directement un exécutable (représenté par la boîte en pointillé).

Tout comme avec la liaison réflexive à l'exécution, il est possible, avec la liaison réflexive à la compilation, de modifier la sémantique d'un langage (Open C++ permet, par exemple, de modifier la sémantique du langage C++ en « ouvrant » l'analyseur syntaxique). Cependant, cette modification

aura lieu lors de la compilation et non lors de l'exécution. Ceci implique, à première vue, que lors de l'exécution il sera impossible d'adapter dynamiquement le système à une nouvelle situation ce qui peut, dans certains cas, limiter de manière conséquente l'utilité de la réflexion.

Une troisième liaison réflexive, apparue récemment, est la *liaison réflexive au chargement de classes*. Elle a été mise en œuvre pour la première fois [4] par Javassist [Chi98]. Dans cette liaison réflexive, le métaobjet n'existe que durant la phase de chargement des classes (il n'est donc présent ni lors de la compilation, ni à l'exécution).

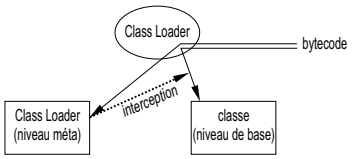


FIG. 7.7 – *Principe de la liaison réflexive au chargement de classes*

Tout comme pour les deux liaisons réflexives précédentes, celle-ci autorise la modification de la sémantique du langage. Pour cela, des métaobjets permettent la modification du code compilé (le *bytecode*) de la classe qui doit être chargée. Cependant, l'adaptation dynamique du système est très limitée. En fait, elle devient, a priori, impossible une fois la classe chargée mais est réalisable lors du chargement de la classe. Pour cela, il suffit de modifier le métaobjet de chargement de classes (figure 7.7).

De la réflexivité structurelle à la compilation vers un MOP à l'exécution

De prime abord, l'utilisation de liaisons réflexives à la compilation ou au chargement de classes implique qu'à l'exécution il ne sera pas possible d'adapter dynamiquement le comportement de l'application en fonction de son contexte comme le permet la liaison réflexive à l'exécution.

En réalité ces deux liaisons réflexives permettent de définir un métaobjet existant à l'exécution et spécifiquement adapté à l'application. En effet, le MOP présent à la compilation ou au chargement de classes peut en générer un autre qui sera présent lors de l'exécution. Grâce à l'analyse qu'il peut effectuer du code de l'application, il peut déterminer (automatiquement, ou en fonction des choix du programmeur) la granularité de la réification qui doit être apportée au MOP présent à l'exécution.

Par exemple si le programmeur souhaite uniquement modifier la sémantique de l'instanciation des objets alors le MOP présent à la compilation ne réifiera que ce concept dans le MOP qu'il définira pour l'exécution. Ainsi, ces liaisons réflexives sont celles qui offrent le meilleur rapport performances / adaptation. En effet, le sur-coût lié au métaobjet n'est pas présent au cours de l'exécution de l'application. De plus, le programmeur du méta-niveau peut définir, lorsqu'il le juge nécessaire, une liaison réflexive à l'exécution optimisée pour le domaine de l'application.

Ces approches, offrant une réflexivité lors des phases de compilation ou de chargement de classes, ne définissent qu'une réflexivité structurelle. Cependant, celle-ci permet, malgré tout, de modifier le comportement général de l'application par une modification de sa structure. Ainsi ces approches offrent généralement des fonctionnalités équivalentes à des approches mettant en œuvre une réflexion comportementale.

EXEMPLE. — Javassist est une librairie de classes offrant une réflexion structurelle dans Java. L'idée essentielle de l'architecture sous-jacente à Javassist est de réaliser la réflexion structurelle par une transformation du bytecode Java lors de son chargement par la machine virtuelle (JVM) [Chi00].

D'autres outils de modifications du bytecode, tels que JOIE [CCK98] ou l'API JavaClass [Dah99], définissent, eux aussi, une réflexion structurelle et offrent des fonctionnalités similaires à celles de Javassist en permettant la modification de la définition d'une classe au chargement de celle-ci. Ces outils, qui mettent en œuvre une réflexion structurelle, procurent, grâce à la liaison réflexive au chargement de classes, des fonctionnalités équivalentes à des outils mettant en œuvre une réflexion comportementale, tels que MetaXa [KG96, GK99] ou Kava [WS99].

7.3 Programmation orientée aspect

La programmation orientée aspect – *Aspect-oriented Programming* (AOP) – est une nouvelle méthodologie de programmation qui permet la séparation modulaire d'*aspects* orthogonaux [MLT +97]. Elle offre une technique permettant d'améliorer sensiblement la séparation des différentes fonctionnalités (*separation of concerns*) d'une application.

Ainsi, le paradigme AOP part du constat suivant : « *Objects have been a great success at facilitating the separation of concerns [...] But objects are limited in their ability to modularize systemic concerns*

4. On peut cependant se demander si l'utilisation de librairies partagées et dynamiquement liées (*dynamic link libraries*) ne peut pas être considérée comme étant une liaison réflexive au chargement de classes, bien que la notion de réflexion et des mécanismes qu'elle entraîne (dont la réification) ne soient pas présents.

that are not localized to a single module's boundaries » [AOP01]. De ce fait, pour les auteurs de la méthodologie AOP, une bonne partie de la complexité et du manque de robustesse des systèmes existants provient du fait que la mise en œuvre des fonctionnalités orthogonales, ou transversales, au langage s'entrelace avec le reste du code de l'application.

L'idée maîtresse du paradigme AOP est basée sur le fait que les mécanismes de modules hiérarchiques (le concept d'objets par exemple dans les langages orientés objet) des langages procéduraux ou orientés objet, bien qu'étant extrêmement utiles, sont intrinsèquement inaptes à encapsuler l'ensemble des fonctionnalités importantes des systèmes complexes. En fait, dans tout système complexe il existe des fonctionnalités qu'il est souhaitable de rendre modulaires (gestion de la concurrence ou de la sécurité par exemple), mais dont la mise en œuvre ne le permet pas. Ceci est le cas lorsque la nature de la modularité de ces fonctionnalités est orthogonale (c'est-à-dire non hiérarchique) à celle de la modularité du reste de la mise en œuvre.

EXEMPLE. — Par exemple, s'assurer qu'un ensemble d'opérations (méthodes) ne s'exécute pas de manière concurrente nécessite normalement de disséminer du code de synchronisation dans chacune de ces opérations.

Une approche orientée aspect permet de spécifier les contraintes de synchronisation dans un morceau de code séparé du reste du code de l'application.

En d'autres termes, l'objectif de la méthodologie AOP est que la modularité d'un système doit refléter la façon de penser de ses concepteurs et non la façon de penser imposée par un langage.

7.3.1 L'aspect, une nouvelle entité modulaire

Pour pallier au problème de modularité orthogonale, la programmation par aspects s'applique aux fonctionnalités qui sont par nature transversales de la même manière que la programmation orientée objet s'applique aux fonctionnalités qui, par nature, sont hiérarchiques : elle fournit des mécanismes, au niveau des langages applicatifs, qui encapsulent explicitement ces structures transversales dans une unité modulaire.

Ceci rend possible la programmation de fonctionnalités transversales de manière modulaire, et, par la même, renforce les bénéfices de la modularité : code plus simple, plus facile à développer et à maintenir, et disposant d'un fort potentiel de réutilisation.

Cette nouvelle unité modulaire est appelée *aspect*. Elle est définie comme suit [KLM+97] :

> « ASPECTS *tend not to be units of the system's functional decomposition, but rather to be properties that affect the performance or semantics of the components in systemic ways.* »

Tout comme les objets, les aspects sont prévus pour être utilisés aussi bien lors de la phase de conception que lors de la mise en œuvre. Au moment de la phase de conception, les concepts d'aspects améliorent le niveau d'abstraction apporté aux fonctionnalités orthogonales (les aspects) en les définissant sous la forme d'entités propres. Lors de la phase de mise en œuvre, les concepts d'aspects permettent de programmer directement en termes d'aspects, de manière identique aux concepts d'objets qui permettent de programmer directement en termes d'objets.

Pour améliorer l'expression de ces fonctionnalités transversales, la programmation par aspects utilise un langage applicatif [5] et des langages d'aspects. Le langage applicatif sert à décrire les fonctionnalités de base du système. Les langages d'aspects [6] permettent de décrire les différentes propriétés orthogonales (les fonctionnalités transversales).

Ainsi un aspect rend modulaire le code vis-à-vis d'une fonctionnalité transversale et définit comment le code décrivant cette fonctionnalité (ce code est appelé programme d'aspect) doit être intégrée au code de l'application (le comportement de base). En fait, la combinaison du comportement de base de l'application avec ceux définis par les programmes d'aspects est réalisée par un « tisseur d'aspects » (*aspect weaver*) qui construit l'application finale (c'est-à-dire le binaire exécutable).

EXEMPLE. — Les interactions, et plus particulièrement les comportements réactifs définis par les interactions, sont des fonctionnalités transversales des concepts de classes et d'objets et leur définition s'unifie parfaitement à celle d'un aspect. Le langage *Interaction Specification Language* (ISL) défini par le modèle à interactions distribuées peut donc être considéré comme un langage d'aspect.

---

5. Ce langage est nommé *component language* par les auteurs de [KLM+97] mais, pour éviter toute confusion avec le paradigme des langages à composants, nous traduirons ce terme par *langage applicatif*.

6. Les auteurs de [KLM+97] proposent la création d'un programme distinct pour chaque aspect, et ce avec le langage de leur choix (c'est-à-dire celui répondant le mieux aux caractéristiques de l'aspect). Ils nomment ces langages des langages d'aspects.

7.3.2 Points d'ancrages et « tissage » des aspects

Les « tisseurs d'aspects » (*aspect weavers*) doivent combiner le programme décrivant l'application avec ceux décrivant les aspects afin de générer l'application finale (par exemple un binaire exécutable).

Ceci est réalisé grâce au concept de points d'ancrages (*join point*), qui sont des éléments de la sémantique du langage applicatif avec lesquels les programmes d'aspects se coordonnent (par composition).

Il est à noter que les points d'ancrages ne sont pas nécessairement des constructions explicites du langage applicatif. Ils peuvent être des éléments (clairement identifiables mais implicites) de la sémantique du programme applicatif. De plus, la description des points d'ancrages n'est pas forcément réalisée dans le même langage que le langage applicatif.

Les tisseurs d'aspects fonctionnent selon le principe suivant : ils génèrent une représentation sous forme de points d'ancrages du programme applicatif [7], puis exécutent (ou compilent) les programmes d'aspects en respectant cette représentation (figure 7.8) et fournissent une nouvelle représentation du programme applicatif.

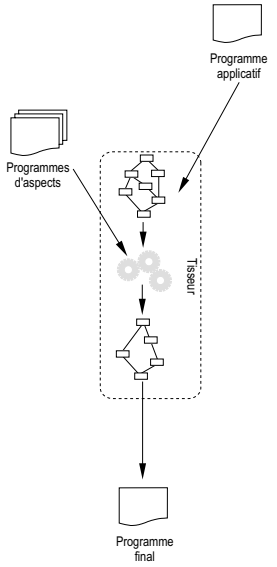


FIG. 7.8 – *Principe du « tissage » des aspects*

La représentation par des points d'ancrage peut être réalisée :

- Lors de la compilation, à l'aide d'un pré-compilateur :
  Dans ce cas, le programme applicatif peut être transformé au niveau du code source en un nouveau programme applicatif prenant en compte les programmes d'aspects et pouvant être compilé par le compilateur du langage applicatif. Le tisseur d'aspects étant alors le pré-compilateur.
- À l'exécution, à l'aide d'un langage applicatif réflexif :
  Dans ce cas, il est possible de mettre en œuvre le langage d'aspect sous la forme d'un programme de méta-niveau, appelé à chaque fois qu'un point d'ancrage est atteint. Le tisseur d'aspects étant, dans ce cas, l'interpréteur du langage réflexif.

EXEMPLE. — Plusieurs systèmes basés sur la programmation par aspects ont défini différents types de points d'ancrage. Par exemple, dans le système de traitement d'images *RG* [MKL97] les points d'ancrage sont les nœuds du graphe de flot de données de l'application. Dans le système de programmation distribuée *D* [LK97], les points d'ancrage correspondent aux corps des méthodes.

Dans AspectJ™ [LK99], les points d'ancrage d'un objet sont les nœuds du graphe d'exécution des appels à cet objet ; ils sont définis dynamiquement au fur et à mesure de l'exécution du programme. Ces nœuds incluent notamment les moments de l'exécution du programme ou un objet reçoit un appel à une méthode et ceux ou un attribut d'un objet est référencé.

Dans notre modèle à interactions distribuées, les comportements réactifs sont déclenchés lors de l'envoi de messages aux objets interagissants. Ainsi les points d'ancrage du langage d'aspect ISL sont les envois de messages.

7.4 Programmation par aspects et réflexivité

La programmation par aspects est fortement liée à la réflexivité et aux protocoles à métaobjets. Comme cela a été présenté dans le paragraphe 7.2, un système réflexif se compose d'un programme au niveau de base (équivalent du programme applicatif de AOP) et d'un, ou plusieurs, programmes de niveaux méta (équivalents des programmes d'aspects de AOP) qui fournissent un contrôle sur la sémantique et la mise en œuvre du niveau de base.

Le méta-niveau fournit un ensemble de vues sur le système qui ne peuvent pas être obtenues par le programme du niveau de base (c'est-à-dire l'application elle-même), telles que la pile complète d'exécution, ou l'ensemble de tous les appels sur tous les objets d'une classe donnée. Ces vues sont orthogonales et transversales au programme du niveau de base. Ainsi, en terme de programmation par aspects, les méta programmes correspondent aux programmes d'aspects pour lesquels les points d'ancrages sont les « poignées » (*hooks*) proposées par le système réflexif.

D'ailleurs, lors du prototypage de leur systèmes à aspects, les auteurs de la programmation par aspects ont souvent utilisé un langage applicatif réflexif et défini un protocole à métaobjets permettant de décrire les programmes d'aspects. Par exemple, l'article [MKL97] décrit le système de traitement

7. Cette génération sous la forme de points d'ancrage est similaire à celle d'un compilateur qui génère une représentation sous forme d'un arbre de syntaxe abstraite d'un programme.

d'images *RG* à l'aide de CLOS et montre comment ce système peut être mis en œuvre de manière efficace grâce au MOP de CLOS. De même, AspectJ™ procure aux programmes d'aspects une information réflexive rudimentaire concernant le point d'ancrage en cours (par exemple la signature de la méthode où se trouve le point d'ancrage) [KHH+01].

Bien évidemment, malgré ces similitudes, les programmations réflexive et par aspects ne sont pas identiques. En effet, avec la programmation par aspects, chaque aspect peut disposer de son propre langage d'aspects (le langage le mieux adapté pour décrire les programmes d'aspects) ; avec la programmation réflexive, le méta programme est généralement décrit dans le même langage que le programme de base. De plus, dans la plupart des cas, les langages réflexifs offrent une interface généraliste du méta-langage (cf. figure 7.1). Celle-ci permet de modifier la sémantique des principaux concepts du langage alors que chaque aspect n'offre qu'une interface spécialisée permettant de ne modifier la sémantique que d'un élément du langage.

7.5 Conclusion

Grâce à la programmation réflexive ou à la programmation orientée aspect les applications peuvent évoluer en fonction du contexte dans lequel elles sont exécutées. De même, le langage de développement peut lui-même être adapté et offrir de nouvelles fonctionnalités qui seront alors utilisées pour décrire les applications.

Dans le contexte de cette thèse, l'adaptation du système est indispensable afin de permettre la définition et l'instanciation dynamiques des interactions. Cependant, les langages compilés et fortement typés (qui constituent l'environnement choisi) limitent grandement cette adaptation. Nous avons montré, dans ce chapitre, comment la réflexivité et la programmation par aspects nous permettent d'outrepasser ces barrières.

Nous présentons, dans le chapitre suivant, notre proposition d'architecture structurelle et fonctionnelle pour la mise en œuvre de notre modèle à interactions distribuées. Nous ne nous sommes cependant pas contentés de définir une mise en œuvre du modèle à interactions distribuées, mais nous avons fait en sorte que cette mise en œuvre soit adaptable et évolutive afin qu'elle puisse prendre en compte de nouveaux domaines d'utilisations des interactions.

Chapitre 8

Un modèle d'architecture

> « *Work on metaobject protocols suggest a new view, in which abstractions expose their implementation, but do so in a way that makes a principled division between the functionnality they provide and the underlying implementation.* » [Kic92]

Nous décrivons, dans ce chapitre, la manière dont les interactions et les schémas d'interactions sont représentés dans notre architecture. De cette représentation, nous définissons un ensemble de classes (et métaclasses) définissant les fondations de notre architecture (noyau minimal). Certaines de ces classes ou métaclasses ont déjà été présentées lors de la définition formelle du modèle. De ces fondations, nous définissons le protocole, existant à l'exécution, permettant la gestion des interactions.

8.1 Description du modèle

Le fait d'intégrer notre modèle à interactions distribuées dans un langage à classes en utilisant un langage compilé et fortement typé comme C++ [Str91] ou Java [GJS96] nous amène, comme nous l'avons vu au chapitre 7, à utiliser un MOP. Ainsi, à la manière du MOP d'Open C++ [Chi95], les métaobjets sont des classes à partir desquelles les comportements initiaux sont spécifiés et peuvent être spécialisés. Ce sont eux qui définissent l'architecture du MOP associé à notre modèle.

8.1.1 Un noyau minimal de sept classes

La figure 8.1 présente le noyau minimal permettant de mettre en œuvre le modèle à interactions distribuées dans un langage à objets basé sur un modèle à objets définissant la notion de métaclasse. L'architecture de ce noyau définit sept nouvelles classes (dont quatre métaclasses) : `MetaObject`, `MetaScheme`, `MetaRule`, `MetaReactive`, `Scheme`, `Rule` et `Reactive` sur lesquelles repose la mise en œuvre du modèle à interactions distribuées et son architecture à métaobjets (MOP).

Ce noyau contient les sept classes (dont quatre métaclasses) suivantes :

`MetaObject` : cette métaclasse définit le comportement permettant la prise en compte des interactions par les objets. En effet, et de manière similaire aux travaux de [Mae87a, CM93], un métaobjet (et un seul) gérant les interactions au niveau des objets (dont notamment le contrôle de l'envoi de messages) est associé à un objet.
Ce métaobjet permet la prise en compte, par le système, des extensions apportées aux objets par le modèle à interactions distribuées. Ce métaobjet a aussi en charge le contrôle de l'envoi de messages et l'exécution des comportements réactifs associés aux méthodes de l'objet qu'il contrôle.

`MetaScheme` : cette métaclasse définit les comportements communs à tous les schémas d'interactions. Ainsi, lorsqu'un schéma d'interactions est défini, sa classe est instance, directe ou indirecte, de la classe `MetaScheme`. C'est donc la métaclasse de toutes les interactions. Elle correspond à la métaclasse nommée $c_{metascheme}$ dans la définition formelle du modèle.

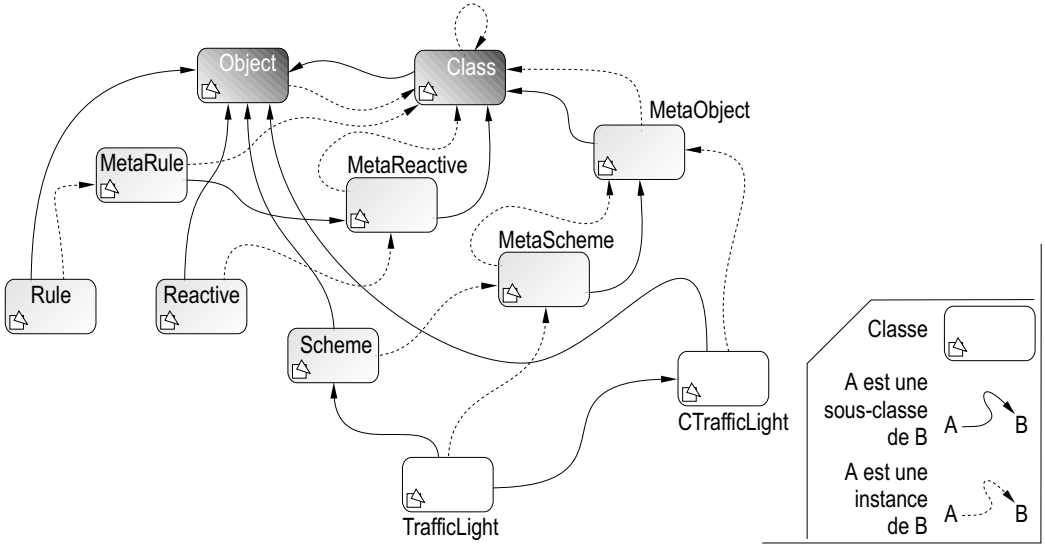


FIG. 8.1 – *Noyau minimal du modèle à interactions distribuées*

`MetaReactive:` cette métaclasse définit les comportements communs aux règles d'interaction et aux comportements réactifs. Elle empêche la déclaration d'interactions sur ses instances. Elle correspond à la métaclasse nommée $c_{metareactive}$ dans la définition formelle du modèle.

`MetaRule:` cette métaclasse définit les comportements communs aux règles d'interaction. Elle correspond à la métaclasse nommée $c_{metarule}$ dans la définition formelle du modèle.

`Scheme:` cette classe définit les comportements communs à toutes les interactions. Ainsi, lorsqu'une interaction est définie, la classe la représentant (son schéma d'interactions) hérite directement ou indirectement de la classe `Scheme`. La classe `Scheme` est une classe abstraite, instance de `MetaScheme`. Elle correspond à la classe nommée c_{scheme} dans la définition formelle du modèle.

`Rule:` cette classe définit les comportements communs à toutes les règles réactives. Elle définit notamment les attributs d'instances présents dans toutes les règles d'interaction. Elle est une instance de `MetaRule`. Elle correspond à la classe nommée c_{rule} dans la définition formelle du modèle.

`Reactive:` cette classe définit les comportements communs à tous les comportements réactifs. Elle est une classe abstraite instance de `MetaRule`. Elle correspond à la classe nommée $c_{reactive}$ dans la définition formelle du modèle.

Les sept classes représentées en grisées sur la figure 8.1 sont les sept classes définies par l'architecture du modèle à interactions distribuées. Les deux classes comportant un fond noir en dégradé de gris (classes `Object` et `Class`) sont les métaclasses du noyau du modèle à objets.

La figure 8.1 présente également un exemple de définition d'un schéma d'interactions (dont la classe est `TrafficLight`) et dévoile les liens (d'instanciation et d'héritage) entre ce schéma d'interactions et les classes de l'architecture du modèle à interactions distribuées.

EXEMPLE. — Dans la figure 8.2, le schéma d'interactions `TrafficLight` est une classe héritant des classes `CTrafficLight` et `Scheme`. Il est une instance de la classe `MetaScheme`. Cette dernière déclare notamment deux attributs, `Behavior` et `Participants`, correspondant respectivement aux attributs *R* et *P* définis par le modèle à interactions distribuées (se reporter au paragraphe 5.4 pour une description de ces attributs).

8.1.2 Description fonctionnelle des schémas d'interactions

Lors de la définition d'un schéma d'interactions, une classe le représentant est créée. Cette classe est, par définition du concept de schéma d'interactions, une instance de la métaclasse `MetaScheme`. Cette métaclasse définit, pour chaque schéma d'interactions, deux attributs (des variables de classes) correspondant à la liste des comportements réactifs définis dans le schéma d'interactions et à la liste des classes des participants à l'interaction.

La classe `MetaScheme` définit aussi l'héritage entre les schémas d'interactions qui a lieu lors de l'instanciation d'un schéma d'interactions. Or, l'héritage entre schémas d'interactions s'applique aux composants d'un schéma d'interactions, c'est-à-dire sa structure et son ensemble de comportements réactifs. L'héritage proposé par le modèle à interactions est spécifique pour les comportements réactifs.

De ce fait, la classe `MetaScheme` s'occupe de l'héritage des comportements réactifs. Cette gestion de l'héritage des comportements réactifs est définie par la classe `MetaScheme` dans sa méthode, nommée

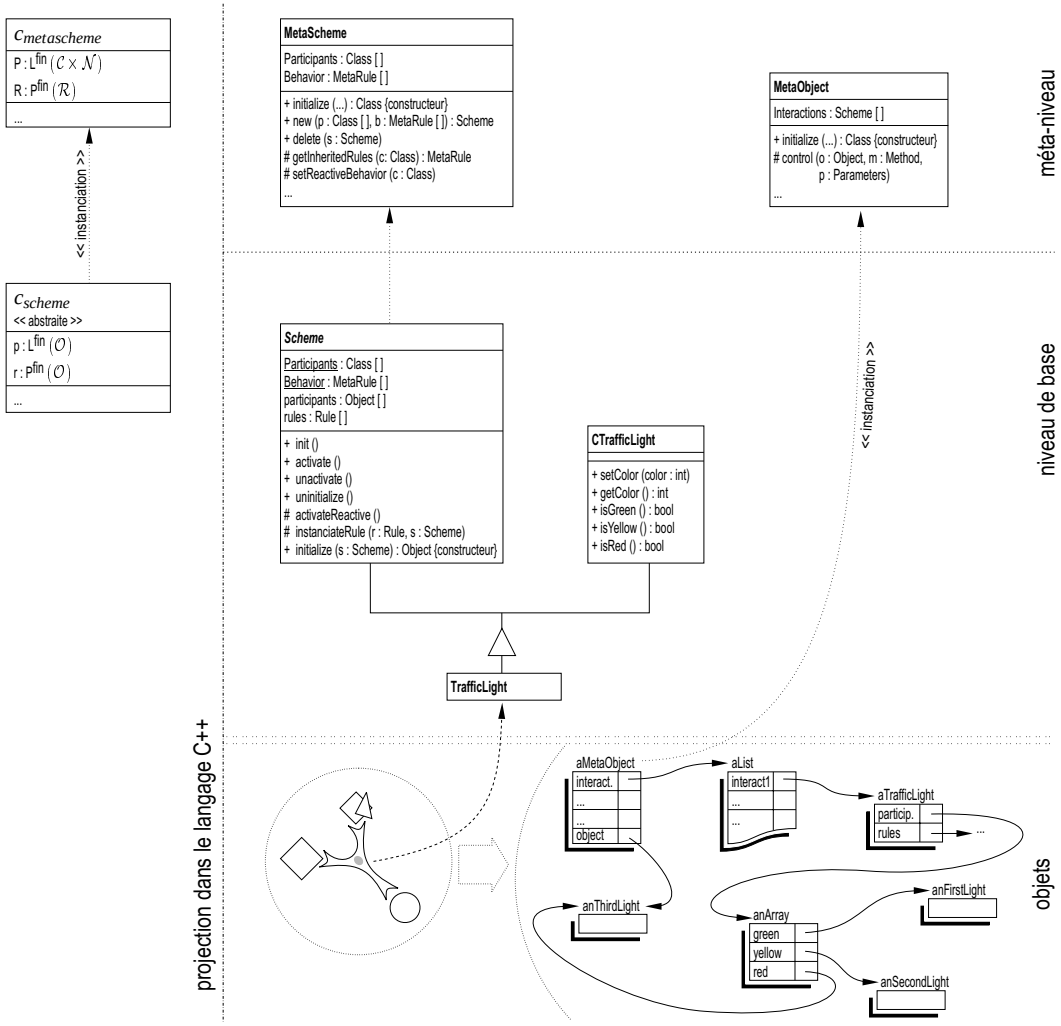


FIG. 8.2 – *Projection du modèle à interactions distribuées dans le langage C++*

`initialize`, décrivant le comportement de la création de ses instances, les schémas d'interactions. Ainsi, comme le montre l'algorithme de figure 8.3 page suivante, la méthode `initialize` de la classe `MetaScheme` invoque les méthodes ci-dessous (définies dans la classe `MetaScheme`).

`setReactiveBehavior`: cette méthode définit l'ensemble des comportements réactifs associés à un schéma d'interactions. Son rôle est donc d'initialiser la valeur de l'attribut *R* (nommé `behavior` par la suite) des schémas d'interactions.
Pour ce faire, elle invoque la méthode `getInheritedRules`. Puis elle « combine » (par application de la méthode `mergeRules`) les règles d'interaction renvoyées par la méthode `getInheritedRules` avec celles définies dans le schéma d'interactions selon le principe d'héritage des comportements réactifs (décrit en 5.6).

`getInheritedRules`: cette méthode renvoie la liste des règles d'interaction définies dans les super-schémas d'interactions en ayant pris soin de renommer les noms des participants si nécessaire en invoquant la méthode `renameParticipantsNames`.

`renameParticipantsNames`: cette méthode gère le renommage des noms des participants afin qu'une règle d'interaction provenant d'un super-schéma d'interactions utilise les mêmes noms des participants que les règles d'interaction issus du schéma d'interactions.

`mergeRules`: cette méthode met en œuvre la sémantique de fusion comportementale d'un ensemble de règles d'interaction.

Ces comportements, définis par la classe `MetaScheme`, permettent de spécialiser le mécanisme d'héritage des schémas d'interactions. Le comportement définit par la métaclasse `MetaScheme` consiste à initialiser les valeurs des deux attributs définis par la classe `MetaScheme`. L'initialisation de l'attribut `behavior` consiste à mettre en œuvre le mécanisme d'héritage des règles d'interaction défini par le modèle à interactions distribuées.

fonction *MetaScheme::initialize* (...) : *Class*;
début
    c:=*Class:initialize* ();
    setReactiveBehavior (*c*);
    c.participants:= . . .;
    retourner *c*
fin

procédure *MetaScheme::setReactiveBehavior* (*Class c*);
début
    initialiser *c.behavior* à l'ensemble vide;
    hr:=*getInheritedRules* (*c*);
    pour chaque règle réactive *r* définie dans le schéma d'interactions **faire**
        si il existe une règle réactive *r′* dans *hr*
            telle que le message déclencheur de *r′* soit le même que celui de *r* **alors**
                si *r* contient le mot-clef `super` **alors début**
                    déclarer *mr* comme un ensemble de règles réactives et l'initialiser
                    à l'ensemble vide;
                    pour chaque règle réactive *ri* de *hr*
                        telle que le message déclencheur de *ri* soit le même que celui de *r* **faire**
                        ajouter *ri* à *mr*
                    r″:=*mergeRules* (*mr*);
                    soit *sr* le résultat de la substitution du mot-clef `super` par le comportement réactif
                        de *r″* dans *r*;
                    ajouter *sr* à *c.behavior*
                fin
                sinon
                    ajouter *r* à *c.behavior*
        sinon
            ajouter *r* à *c.behavior*
fin

fonction *MetaScheme::getInheritedRules* (*Class c*) : MetaRule;
début
    déclarer *hr* comme un ensemble de règles réactives et l'initialiser à l'ensemble vide;
    pour chaque super-schéma d'interactions *s* **faire début**
        soit *sr* l'ensemble des règles réactives définies dans *s*;
        pour chaque règle d'interaction *r* de *sr* **faire**
            remplacer *r* par *renameParticipantsNames* (*r*) dans *sr′*
        ajouter *sr′* à *hr*
    fin
    retourner *hr*
fin

FIG. 8.3 – *Algorithme de définition d'un schéma d'interactions*

EXEMPLE. — Soit les deux schémas d'interactions suivants :

```
interaction SimpleDrinkMachine (PushButton button, Container container)
{
  button.Push () -> button.Push () ; container.RemoveOne ()
}

interaction DrinkMachine (PushButton push, Container container, Money money)
    extend SimpleDrinkMachine (push, container)
{
  push.Push () -> if money.EnoughMoney () then super () ; money.Debit () else delegate money.Refund () endif
}
```

Lors de la définition du schéma d'interactions `DrinkMachine` les noms des participants de la règle d'interaction du schéma d'interactions `SimpleDrinkMachine` vont être renommés (ainsi le nom du bouton poussoir `button` va être renommé en `push`). Ensuite, cette règle d'interaction va être introduite dans la règle d'interaction du schéma d'interactions `DrinkMachine` en lieu et place du mot-clef `super` (ici la fusion comportementale ne fait rien puisqu'il n'y a qu'une seule règle d'interaction).

Une fois le schéma d'interactions défini, celui-ci peut être instancié autant de fois que nécessaire entre différents ensembles d'objets.

8.2 Description des comportements réactifs

8.2.1 Les comportements réactifs mis en œuvre par des opérateurs

L'exécution d'un comportement réactif nécessite différentes opérations : envoyer un message, exécuter de manière concurrente deux comportements réactifs (opérateur réactif concurrentiel), déléguer un message, attendre la fin de l'exécution d'un message, etc.

Ces différentes opérations, de nature très différentes, sont associées à un opérateur réactif afin de permettre la définition de nouveaux opérateurs réactifs. Ceci nous a incité à ne définir dans les classes `MetaScheme` et `Scheme` aucun comportement directement associé à un opérateur réactif particulier. La gestion de ce type de comportement est déléguée à l'opérateur réactif lui-même. Pour ce faire, nous avons réifié tous les opérateurs réactifs.

Ainsi, cette prise en compte de la possibilité d'ajout de nouveaux opérateurs réactifs et de la différence d'exécution nous a conduit à étendre l'architecture de la figure 8.1 par l'ajout de nouvelles classes. Nous obtenons l'architecture de la figure 8.4.

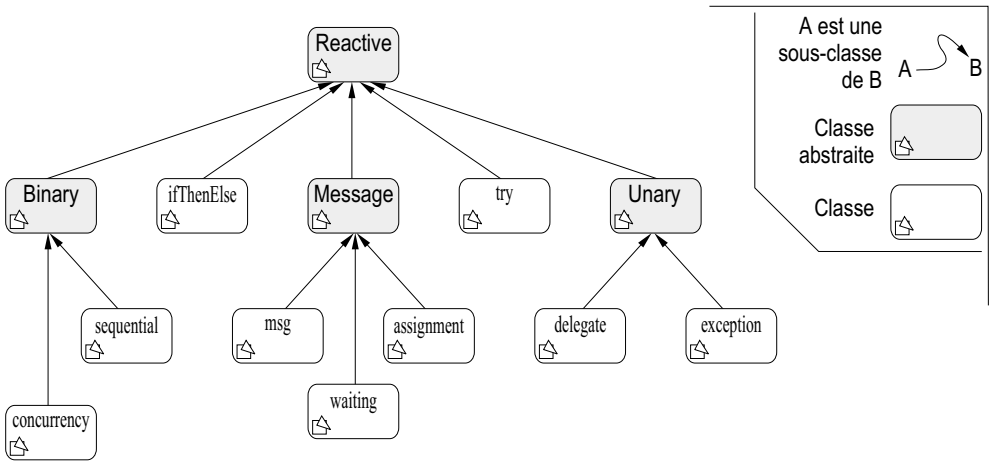


FIG. 8.4 – *Architecture liée aux opérateurs réactifs*

Cette architecture définit une classe spécifique pour chacun des opérateurs réactifs définis par le modèle à interactions distribuées. Elle introduit les trois classes abstraites suivantes (en plus de la classe `Reactive` déjà décrite ci-dessus) :

`Unary` : cette classe décrit les attributs et les comportements communs aux opérateurs réactifs unaires.

`Binary` : cette classe décrit les attributs et les comportements communs aux opérateurs réactifs binaires. Elle contient notamment les comportements décrivant la sémantique d'exécution d'un comportement réactif et les sémantiques d'exécution de deux comportements réactifs de manière séquentielle ou concurrentielle.

`Message` : cette classe décrit les attributs et les comportements communs aux opérateurs réactifs impliquant un envoi de messages. Elle contient notamment la sémantique de l'envoi de messages utilisée par chacune de ses sous-classes.

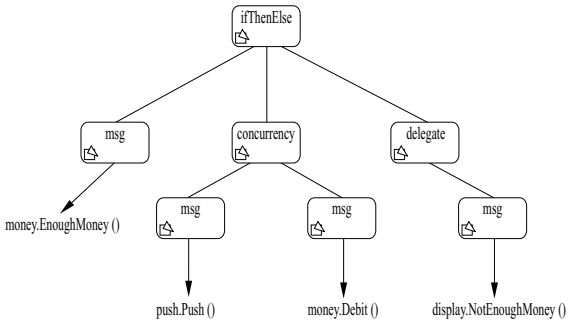
Comme cela a été vu dans le paragraphe 5.3.2, les différents opérateurs réactifs ont à leur charge l'exécution du comportement réactif qu'ils définissent ainsi que la sémantique de la fusion comportementale qui leur est associée.

De ce fait, chacune des classes définies ci-dessus dispose d'une méthode, nommée `execute` et héritée de la classe `Reactive`, permettant l'exécution de leur comportement réactif. Elles définissent également un ensemble de méthodes permettant, grâce au polymorphisme, la fusion comportementale de leur comportement réactif avec un autre comportement réactif. Cependant, afin de permettre l'ajout de nouveaux opérateurs réactifs et de permettre leur fusion comportementale avec les autres opérateurs réactifs, lorsqu'aucune méthode décrivant la sémantique de fusion comportementale appropriée ne peut être appliquée sur l'un des opérateurs réactifs, la fusion comportementale est déléguée à l'autre opérateur réactif.

EXEMPLE. — Soit la règle d'interaction suivante :

```
push.Push () -> if money.EnoughMoney () then push.Push () // money.Debit ()
                                          else delegate display.NotEnoughMoney () endif
```

La structure du comportement réactif de cette règle d'interaction est la suivante :



On peut noter que le comportement réactif de la règle d'interaction est décrit par une arborescence d'opérateurs réactifs.

8.2.2 Définition de nouveaux opérateurs réactifs

L'architecture proposée pour les opérateurs réactifs permet la définition de nouvelles sémantiques de comportement réactif et donc, par conséquent, l'introduction de nouveaux opérateurs réactifs.

En effet, introduire un nouvel opérateur réactif implique de définir une nouvelle classe héritant, directement ou indirectement, de la classe `Reactive`, de redéfinir la méthode `execute` afin qu'elle reflète la sémantique d'exécution du nouvel opérateur réactif, puis de définir un ensemble de méthodes permettant de réaliser la fusion comportementale de cet opérateur réactif avec les autres opérateurs réactifs.

EXEMPLE. — Supposons qu'un programmeur définisse très souvent des interactions dont le comportement réactif ne doit être déclenché que lorsque deux messages m_1 et m_2 ont été invoqués dans cet ordre. Avec les opérateurs réactifs que nous avons définis dans le langage ISL, il écrit une telle sémantique comme suit :

```
interaction anInteraction (aFirstMember member1, aSecondMember member2) implement SchemeWithFlag
{
  member1.method1 (int x) -> member1.method1 (x) // this.SetFlag ('true'),
  member2.method2 (string s) -> if (this.IsFlagSet ()) then this.SetFlag ('false') ; reaction else member2.method2. (s)
}
```

Cette solution impose donc la définition d'une classe particulière pour les schémas d'interactions (pour décrire la gestion de la valeur du drapeau déclencheur du comportement réactif à exécuter lors de la réception des deux messages m_1 et m_2).

Par conséquent, notre programmeur souhaite définir un, ou plusieurs, nouveaux opérateurs réactifs permettant d'éviter la définition d'une classe spécifique. Il étend la syntaxe concrète du langage ISL de manière à pouvoir écrire l'interaction ci-dessus comme suit :

```
interaction anInteraction (aFirstMember member1, aSecondMember member2)
{
  member1.method1 (int) & member2.method2 (string s) -> reaction
}
```

Ce code ISL étendu va être « transformé » comme suit (les deux nouveaux opérateurs réactifs sont en gras) :

```
interaction anInteraction (aFirstMember member1, aSecondMember member2)
{
  member1.method1 (int param1) -> flag member1.method1 (param1)
  member2.method2 (string s) -> flagOnly (member1.method1 (int), reaction)
}
```

L'opérateur réactif `flag` dispose de la même sémantique exécutoire que l'opérateur réactif `msg` : il consiste à exécuter le message `member1.method1 (param1)`. Par conséquent, les règles de fusion comportementale de l'opérateur réactif `flag` sont basées sur celle de l'opérateur `msg`. De même, la classe décrivant l'opérateur `flag` hérite de celle de l'opérateur `msg`.
L'opérateur réactif `flagOnly` a une sémantique similaire à l'opérateur réactif `ifThenElse` puisqu'il n'exécute le comportement réactif `reaction` que si le message `member1.method1` a été « encapsulé » par l'opérateur réactif `flag`. Dans le cas contraire le message déclencheur est exécuté. Les règles de fusion comportementale de cet opérateur réactif sont basées sur celles de l'opérateur réactif `ifThenElse`.

NOTE. — Avec cette solution le comportement réactif désigné par l'opérateur réactif `flagOnly` n'a pas accès à l'environnement d'exécution du premier message déclencheur (`member1.method2` dans notre exemple). De plus, elle définit une relation d'ordre sur les messages déclencheurs (`member1.method1` puis `member2.method2`).

8.3 Utilisation des comportements réactifs

Nous décrivons maintenant comment les comportements réactifs sont utilisés. L'importance accordée par le modèle à interactions distribuées aux comportements réactifs nous amène, en effet, à apporter, dans ce paragraphe, des précisions sur les fonctionnalités des comportements réactifs et la manière de les exploiter au mieux.

8.3.1 Propagation de l'exécution des comportements réactifs

Chaque interaction définit un contrôle des messages sur les objets participants. Or un objet peut être le participant de plusieurs interactions. Ainsi, les interactions définissent un graphe orienté de comportements réactifs dont les nœuds sont les objets et les arcs des envois de messages. De ce fait, l'exécution d'un comportement réactif peut déclencher, à son tour, l'exécution d'un autre comportement réactif, et ainsi de suite.

EXEMPLE. — Dans l'exemple de l'égaliseur graphique (figure 8.5 de la présente page), l'objet`b1`, une instance de la classe `button`, est un objet libre. Ainsi un envoi du message `enDeplacement` sur cet objet implique une exécution normale de la méthode.

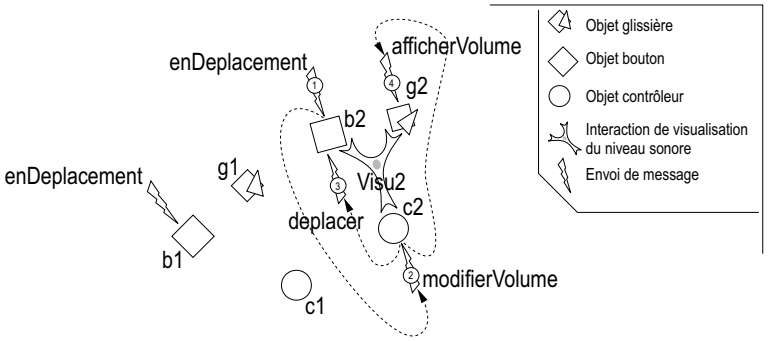


FIG. 8.5 – *Contrôle des envois de messages induit par les comportements réactifs*

Par contre, l'objet `b2`, lui aussi instance de la classe `button`, est lié à l'interaction `Visu2` car une règle d'interaction, dont le message déclencheur est `enDeplacement`, est définie dans l'interaction `Visu2`. Ainsi l'envoi du message `enDeplacement` sur l'objet `b2` implique l'exécution du comportement réactif associé à ce message.

NOTE. — Pour éviter une boucle infinie d'exécution des comportements réactifs, l'invocation du message déclencheur dans le comportement réactif ne provoque pas une propagation du contrôle (c'est-à-dire n'exécute pas à nouveau le comportement réactif associé au message) mais applique la sémantique traditionnelle de l'envoi de messages (c'est-à-dire sans contrôle).

La figure 8.6 montre le graphe de propagation des comportements réactifs pour l'exemple de l'égaliseur graphique. Les arcs du graphe sont les invocations de messages, les nœuds étant les objets du système.

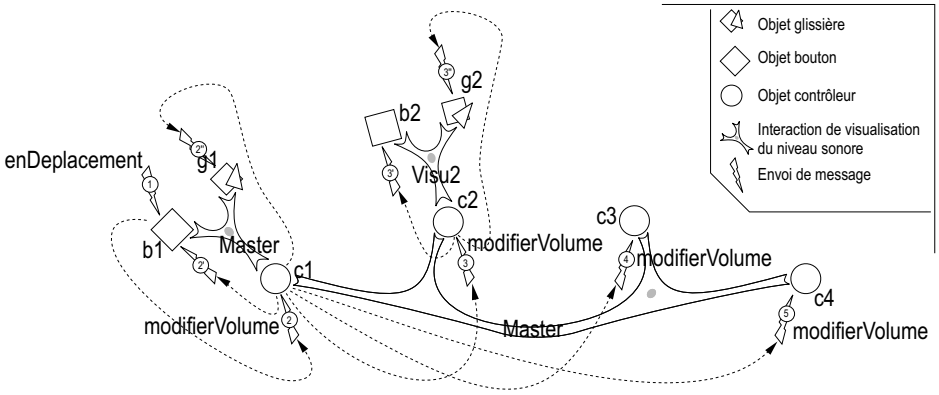


FIG. 8.6 – *Graphe de propagation des comportements réactifs*

8.3.2 Exécution des comportements réactifs

Comme cela a été vu ci-dessus, l'exécution d'un comportement réactif nécessite différentes opérations, de nature très différentes. Jusqu'à présent nous n'avons pas abordé la façon dont est réalisé le contrôle des messages, nous avons juste donné sa sémantique. Nous comblons, en partie, ce manque avec ce paragraphe.

Le contrôle des messages est réalisé par le métaobjet `MetaObject` introduit par le modèle à interactions distribuées. Ce contrôle est implicite et transparent du point de vue de la communication entre objets, c'est-à-dire que les messages sont toujours envoyés aux objets et non à leurs métaobjets (figure 8.7).

Le métaobjet a aussi en charge l'exécution des comportements réactifs, c'est-à-dire la prise en compte de la sémantique du contrôle direct. En effet, l'exécution des comportements réactifs associés à

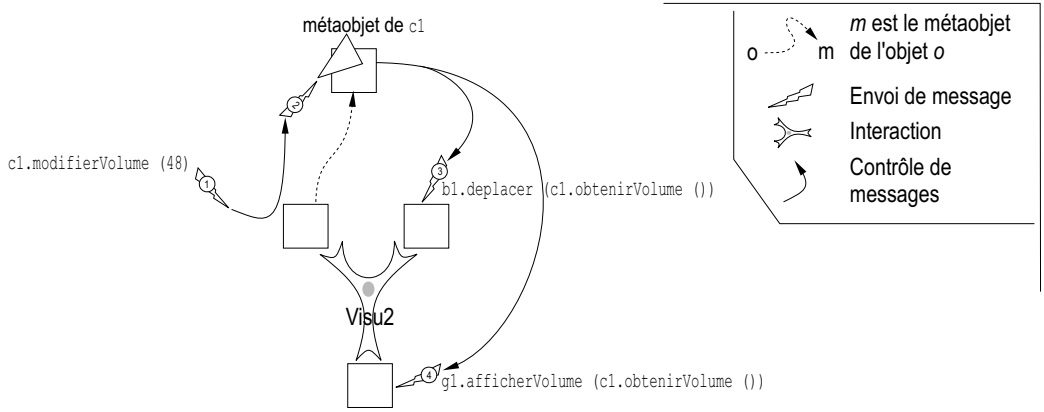


FIG. 8.7 – *Mécanisme du contrôle des messages*

un message est incluse dans la sémantique du contrôle des messages. La figure 8.8 présente l'algorithme du contrôle des messages décrivant la sémantique d'exécution des comportements réactifs. Celle-ci fait appel au mécanisme de fusion comportementale décrite dans le chapitre 6.

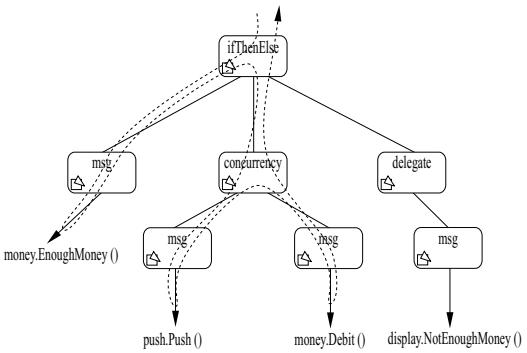
procédure *MetaObject::control* (*Object o*, *Method m*, *Parameters params*);
début
déclarer *cr* comme un ensemble de règles d'interaction et l'initialiser
à l'ensemble vide;
pour toutes les règles d'interaction *r* dont le message déclencheur est le couple (m, o) **faire**
ajouter *r* à *cr*
si l'ensemble *cr* est non vide **alors début**
soit *c* le comportement réactif résultant de la fusion comportementale des
règles d'interaction contenues dans *cr*;
exécuter *c*
fin
sinon
exécuter le message en utilisant la sémantique normale
fin

FIG. 8.8 – *Algorithme d'exécution des comportements réactifs*

EXEMPLE.— Reprenons l'exemple du paragraphe 8.2.1. La figure ci-contre présente le flot d'envoi de messages lors de l'exécution de la règle d'interaction décrite par le comportement réactif lorsque la condition est vérifiée (ligne en pointillés).

Comme l'on peut le voir, l'arbre des opérateurs réactifs décrivant le comportement réactif à exécuter est parcouru en utilisant un algorithme de parcours en profondeur. Ce parcours est, en fait, réalisé par invocation de la méthode `execute` sur l'opérateur réactif `ifThenElse`.

Cette méthode contient la sémantique d'exécution de cet opérateur réactif. Par conséquent, elle invoque la méthode `execute` sur l'opérateur réactif `msg` décrivant la condition puis, en fonction de la valeur de retour de cette exécution, invoque la méthode `execute` soit sur le comportement réactif décrivant la partie `then`, soit sur celui décrivant la partie `else`.

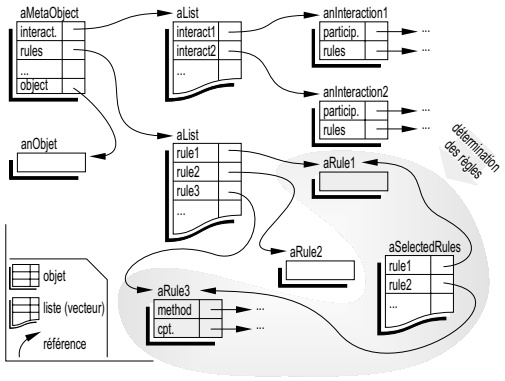


8.3.3 Détermination et fusion des règles d'interaction à exécuter

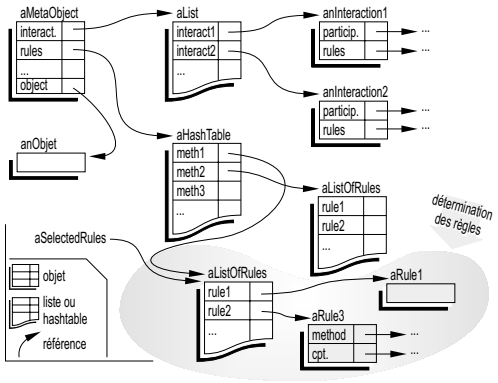
Nous avons vu ci-dessus (figure 8.8) que, lorsqu'un message contrôlé est invoqué, le métaobjet peut consulter l'ensemble des interactions et par conséquent obtenir la liste des comportements réactifs (sous la forme de règles d'interaction) associés au message capturé. Cependant, ceci peut impliquer, chaque fois qu'un message contrôlé est invoqué, un ensemble de communications entre le métaobjet et ses interactions. Or tous ces objets peuvent être situés sur différents sites, et donc une communication entre eux peut nécessiter des transferts via le réseau. De ce fait, les performances associées à la détermination des règles d'interaction à exécuter peuvent être extrêmement mauvaises.

L'une de nos motivations est la minimisation de l'empreinte de notre support des interactions sur l'application (critère C2.4). Par conséquent, la dégradation des performances en termes de temps d'exécution doit être la plus faible possible. De ce fait, pour éviter les communications réseaux entre les métaobjets et les interactions lors de la capture d'un message, les métaobjets contiennent, dans une liste, une copie des comportements réactifs qui leurs sont associés (figure 8.9a).

Ainsi, déterminer les comportements réactifs à exécuter lors de la capture d'un message revient à parcourir cette liste (les règles d'interaction dont le fond est gris sont celles qui ont été sélectionnées comme devant être exécutées suite au parcours de la liste).



8.9a – Détermination des règles d'interaction à l'aide d'une liste



8.9b – Détermination des règles d'interaction à l'aide d'une table de correspondances

FIG. 8.9 – *Détermination des règles d'interaction à exécuter*

Afin d'améliorer encore les performances, les comportements réactifs peuvent être stockés dans le métaobjet sous la forme d'une table de correspondances (*hashtable*) dont la clef est la signature de la méthode (figure 8.9b). Dans ce cas, la détermination des règles d'interaction à exécuter est directe. Une fois les règles d'interaction à exécuter déterminées, il faut les combiner avant leur exécution effective (par application de la fusion comportementale).

Fusion des règles d'interaction

La phase de fusion des règles d'interaction consiste à appliquer la fonction de fusion comportementale (décrite dans le chapitre 6) à l'ensemble des règles d'interaction obtenues par l'étape de sélection des règles d'interaction à exécuter (étape décrite ci-dessus).

Les propriétés de commutativité (propriété 6.2 page 92) et d'associativité (propriété 6.3 page 96) de la fonction de fusion comportementale permettent d'appliquer la fonction de fusion comportementale sur la liste des règles d'interaction à fusionner sans avoir à se préoccuper de la manière dont la fonction de fusion comportementale est appliquée (pas de notion d'ordre).

L'application de la fonction de fusion comportementale peut avoir lieu soit lors de l'activation des comportements réactifs, soit lors de leur exécution. Pour des raisons d'efficacité, nous avons choisi de mettre en œuvre cette dernière solution.

Cependant, ceci implique d'appliquer la fonction de fusion comportementale sur les règles d'interaction à exécuter à chaque fois que le message déclenchant ces règles d'interaction a été invoqué (par envoi d'un message). Or, cette opération est complexe et est très gourmande en temps d'exécution (elle est proportionnelle au nombre de règles d'interaction à fusionner et au niveau d'imbrication des opérateurs réactifs pour chacune des règles d'interaction à fusionner).

De par la définition de la fonction de fusion comportementale, si entre deux invocations d'une méthode, aucune règle d'interaction n'a été supprimée ou ajoutée (par ajout ou suppression d'interactions), le résultat de la fonction de fusion comportementale sera le même (puisque l'ensemble des règles d'interaction à fusionner est le même). Il est donc possible de stocker dans un cache le résultat de la précédente application de la fonction de fusion comportementale et d'utiliser, si l'ensemble des règles d'interaction n'a pas été modifié, le contenu de ce cache lors de la prochaine invocation de la méthode. Ainsi, la fonction de fusion comportementale n'est appliquée que lorsque cela est strictement nécessaire.

Il est possible d'optimiser encore un peu la fusion des règles d'interaction. En effet, lors de l'ajout d'une règle d'interaction à une méthode, il n'est pas nécessaire de re-fusionner les autres règles d'interaction associées à ce même message, puisque le résultat de cette fusion est contenu dans le cache. Dans ce cas, il suffit de fusionner le résultat du cache avec la nouvelle règle d'interaction. En fait, il est nécessaire de refusionner toutes les règles d'interaction que si une, ou plusieurs, règles d'interaction sont supprimées.

8.3.4 Unification des participants et des arguments des méthodes

Une règle d'interaction est l'association d'un message et d'un comportement réactif. Elle est spécifiée en termes de l'interface des objets participants. Les appels de méthodes dans les comportements réactifs sont définis à l'aide de paramètres formels (c'est-à-dire que les paramètres des méthodes sont représentés par des noms).

Unification des participants

Lors de l'instanciation d'un schéma d'interactions les noms des participants et les identifiants des participants sont unifiés dans les règles d'interaction. Les paramètres des méthodes restant des paramètres formels. Nous appelons cette unification l'unification des participants.

EXEMPLE. — Soit le schéma d'interactions suivant, issu d'un jeu de bataille navale (paragraphe 4.5.1) :

```
interaction NavyBattle (Array array, Graphic graphic) implement ConverteCoord
{
  array.setState (int coord, int state) -> array.setState (coord, state) //
                                           graphic.display (this.getX (coord), this.getY (coord), state)
}
```

Soit l'instanciation de ce schéma d'interactions comme suit :

```
theNavyBattle = new NavyBattle (theArray, theGraphic);
```

L'unification des participants produit la règle d'interaction suivante :

```
theArray.setState (int coord, int state) -> theArray.setState (coord, state) //
                                              theGraphic.display (theNavyBattle.getX (coord),
                                                                  theNavyBattle.getY (coord), state)
```

Unification des arguments des méthodes

Lors de l'exécution d'un comportement réactif, les paramètres formels doivent être associés avec les valeurs des arguments afin que les méthodes contenues dans le comportement réactif puissent être invoquées. Ceci peut être réalisé par filtrage (*pattern matching*) car les paramètres formels sont liés aux arguments du message.

EXEMPLE. — Reprenons l'exemple précédent (jeu de bataille navale). Dans la règle d'interaction, définie par le schéma d'interaction `NavyBattle`, le symbole `state` de la liste d'arguments (troisième argument) de la méthode `display` a pour valeur la valeur du second argument lors de l'invocation de la méthode `setState`.

L'unification des paramètres de l'envoi d'un message dans un comportement réactif autorise le concepteur d'un schéma d'interactions à manipuler ces paramètres dans le but de leur appliquer des méthodes ou de les adapter à la signature des méthodes sur lesquelles ils vont être appliqués :

- L'interaction peut convertir les arguments ou en générer de nouveaux. Ceci permet d'exprimer aisément des conversions de types. Ces conversions peuvent être exprimées en tant que méthode propre à l'interaction.
- L'interaction peut également adapter à la fois l'ordre et le nombre des paramètres pour rendre valide la communication entre des objets. Cette adaptation est alors liée à l'interaction.

EXEMPLE. — Reprenons à nouveau l'exemple de la bataille navale. La règle d'interaction suivante permet de convertir la valeur d'un paramètre pour l'adapter aux types des paramètres du message invoqué :

```
array.setState (int coord, int state) -> array.setState (coord, state) //
                                         graphic.display (this.getX (coord), this.getY (coord), state)
```

Dans la règle d'interaction ci-dessus, le nombre des paramètres a été adapté afin que l'invocation de la méthode `display` sur l'objet identifié par le symbole `graphic` soit correcte vis-à-vis de la signature de cette dernière.

Le statut dont dispose les interactions (elles sont des objets pouvant interagir) et leur spécificité implique l'utilisation des deux mots-clefs suivants :

this référence l'objet interaction elle-même (comme `this` en C++ ou Java). En effet une interaction est un objet disposant d'un comportement et des variables propres pouvant être utilisés dans les règles d'interaction. L'introduction de ce mot-clef est donc indispensable. Ce mot-clef est donc unifié avec l'identifiant de l'interaction.

super référence les comportements réactifs définis dans les super-schémas d'interactions. En effet, les comportements réactifs définis dans un schéma d'interactions peuvent être redéfinis, par raffinement, dans un sous-schéma d'interactions. Ce mot-clef permet à un schéma d'interactions de réutiliser les comportements réactifs déjà définis dans un super-schéma d'interactions. Il est donc, lui aussi, indispensable.

8.4 Cycle de vie d'une interaction

En nous basant sur la description structurelle du modèle, nous abordons dans ce paragraphe une description fonctionnelle de la prise en compte des interactions au travers des différentes étapes qui jalonnent la vie d'une interaction, à savoir son instanciation, son activation, son inhibition et sa destruction.

8.4.1 Instanciation d'un schéma d'interactions

L'instanciation d'une classe est réalisée grâce au concept de constructeur du modèle à objets qui permet de modifier certains mécanismes de création des instances d'une classe (par exemple l'initialisation des valeurs des attributs). Ainsi, nous redéfinissons ce constructeur, dans la classe `Scheme` ou l'une de ses sous-classes, afin de s'assurer de la cohérence de l'interaction et de gérer l'héritage des comportements réactifs.

En effet, la création d'une interaction revêt un caractère particulier lié aux informations supplémentaires nécessaires pour gérer de manière cohérente, et tout au long de la vie de l'interaction, l'exécution des comportements réactifs définis par l'interaction sous la forme de règles d'interaction. De plus, la possibilité d'exprimer plusieurs règles d'interaction ayant pour une même signature de méthode des objets participants différents implique, étant donné une signature de méthode et un objet, d'être en mesure de déterminer la règle d'interaction associée à cette méthode et cet objet. Ceci implique la mise en place d'un mécanisme de filtrage ou d'unification de motifs (*pattern matching*) lors de l'instanciation du schéma d'interactions.

EXEMPLE. — Soit une interaction, instance du schéma d'interactions `TrafficLight` (dont le code ISL est rappelé ci-dessous), déclarée entre les objets `l1`, `l2`, et `l3`. Lorsque l'objet `l1` reçoit le message `on` il faut être capable de déterminer que c'est la règle d'interaction définie à la ligne 6 qui doit être exécutée. Il faut donc être en mesure d'associer au nom`green` (représentant l'un des objets participants) l'objet`l1`.

```
1: interaction TrafficLight (Light green, Light yellow, Light red)
2:   class CTrafficLight
3: {
4:   this.setColor (int color) -> delegate if this.isGreen then green.on ()
                                       else if this.isYellow () then yellow.on ()
                                            else red.on () endif endif,
5:
6:   green.on ()       -> [ yellow.off () // red.off () ] ; green.on (),
7:   yellow.on ()      -> [ green.off () // red.off () ] ; yellow.on (),
8:   red.on ()         -> [ green.off () // yellow.off () ] ; red.on ()
9: }
```

Le constructeur de la classe `Scheme`, noté `initialize`, est redéfini afin d'initialiser les valeurs des deux attributs d'instances qui lui ont été rajoutés par la métaclasse `MetaScheme`. Ainsi il stocke chaque objet participant dans la liste des participants. Il détermine aussi les règles d'interaction contenues dans l'interaction. L'algorithme de l'instanciation des schémas d'interactions est présenté par la figure 8.10.

Ensuite, la méthode `init` définie par la classe `Scheme` est exécutée. Par défaut, cette méthode ne fait rien. Son but est d'offrir un moyen d'exprimer, grâce à un comportement réactif notamment, une action sur les objets participants avant que l'interaction ne soit active dans le système (pour, par exemple, définir un état initial des objets participants).

Il est à noter que, lorsque cette fonction est exécutée, l'objet interaction existe déjà mais qu'il n'est pas encore entièrement construit (en particulier, le comportement des objets participants à l'interaction n'est pas encore modifié par les comportements réactifs définis par l'interaction).

```
fonction Scheme::initialize (Scheme s) : Object;
début
    o:=super::initialize (s);
    pour toute règle réactive r définie dans s  faire
        ajouter instanciateRule (r, s) à o.behavior
    si il existe une règle réactive r associée au message déclencheur this.init  alors
        exécuter le comportement réactif de r
    sinon
        init ()
    activateReactive ();
    retourner o
fin

fonction Scheme::instanciateRule (Rule r, Scheme s);
début
    pour chaque participant p de l'interaction  faire
        remplacer par p dans r chaque occurrence du nom correspondant dans s à p
    retourner r
fin
```

FIG. 8.10 – *Algorithme d'instanciation d'un schéma d'interactions*

Une fois le constructeur exécuté, l'interaction est physiquement construite. Pour qu'elle soit effective, c'est-à-dire pour que les comportements réactifs qu'elle définit soient pris en compte par le système, il suffit de l'activer, grâce à la méthode `activateReactive`.

8.4.2 Activation et inhibition d'une interaction

Au delà de la création effective de l'objet représentant l'interaction, il est nécessaire de pouvoir déterminer clairement le moment exact à partir duquel les comportements réactifs définis par l'interaction sont pris en compte par le système, c'est-à-dire le moment exact ou le comportement des objets participants à l'interaction est modifié du fait de l'interaction.

EXEMPLE. — Dans certains des langages cibles (Java notamment), la création d'un objet est une opération coûteuse en temps d'exécution. Ainsi, minimiser le nombre d'objets créés permet d'améliorer les performances globales de l'application. Or, les interactions sont des objets. Ainsi, lorsqu'un schéma d'interactions est souvent instancié, et détruit, sur un même ensemble d'objets, il est avantageux de réellement l'instancier une unique fois, puis d'activer et désactiver les comportements réactifs qu'il définit (opération moins complexe) de multiples fois.

Prenons l'exemple d'un pare-feu (*firewall*) qui joue aussi le rôle d'un *proxy* Web. À chaque connexion à Internet, ce pare-feu pose une interaction sur la connexion. Or, dans le cas d'un accès à un serveur Web par le protocole HTTP 1.0, le chargement d'une page Web peut impliquer la création de plusieurs connexions (une connexion pour le document Web proprement dit et une connexion pour chacune des images contenues dans ce document). De ce fait, le pare-feu utilise une variante du schéma de conception Poids-Mouche (*Flyweight Design Pattern* [GHJ+95, pp. 195–206]) en disposant d'un « pool » de connexions.

Ceci implique qu'à chaque fois qu'une connexion est sortie du « pool » (afin d'être utilisée) une interaction doit être posée sur cette connexion et, inversement, chaque fois qu'une connexion est remise dans le « pool », son interaction doit être détruite. Une optimisation possible consiste à gérer, en parallèle du « pool » de connexions, un « pool » d'interactions qui seront activées ou désactivées en fonction des besoins évitant, du même coup, des instanciations et destructions répétées de schémas d'interactions sur les mêmes groupes d'objets.

Cette étape d'activation de l'interaction est réalisée par la méthode `activateReactive`. Celle-ci ajoute l'ensemble des comportements réactifs définis par l'interaction dans le système (la figure 8.11 présente l'algorithme d'activation d'une interaction). De plus, sachant que l'ensemble de ces comportements réactifs définissent une sémantique de communication, cette méthode introduit dans le système tous les comportements réactifs de l'interaction de manière atomique vis-à-vis des objets participants à l'interaction. Ceci signifie que durant l'ajout de ces comportements réactifs au système, les messages envoyés aux objets participants à l'interaction sont mis en attente afin d'être traités une fois l'ensemble des comportements réactifs ajoutés. Ceci a pour but de maintenir la cohérence du système.

Le métaobjet d'un objet étant responsable de l'exécution des comportements réactifs déclenchés par cet objet, l'introduction des comportements réactifs d'une interaction consiste à informer, pour chacune des règles d'interaction définies par l'interaction, le métaobjet de l'objet qui va être le déclencheur de la règle d'interaction de l'existence de cette dernière (en lui fournissant une copie du comportement réactif). Ceci est réalisé en invoquant la méthode `registerRule` définie par la classe `MetaObject`. Le protocole complet associé à ces enregistrements des comportements réatifs au sein d'un métaobjet est décrit en Annexe B.

procédure *Scheme::activateReactive* ();
début
    si il existe une règle d'interaction *r* dont le message déclencheur est *this.activate* **alors**
        exécuter le comportement réactif de *r*
    sinon
        activate ()
fin

procédure *Scheme::activate* ();
début
    pour chaque règle d'interaction *r* de l'interaction **faire début**
        soit *mo* le métaobjet associé au message déclencheur de *r*;
        mo.registerRule (*r*)
    fin
fin

FIG. 8.11 – *Algorithme d'activation d'une interaction*

Sémantique des comportements réactifs d'activation

La méthode `activate` est publique (méthode d'interface). Il est donc possible de la contrôler à l'aide de comportements réactifs. Or, cette méthode décrit la sémantique de la création des comportements réactifs dans l'application et est donc étroitement liées à la sémantique d'exécution des comportements réactifs. Ainsi, appliquer la sémantique d'exécution des comportements réactifs aux comportements réactifs définis sur cette méthode pose certains problèmes. Nous allons donc les exposer brièvement puis y apporter une solution.

Supposons qu'une interaction, que nous nommons *i*, définisse, entre autres, les deux règles d'interaction suivantes :

```
R1: stdout.println (string text) -> stdout.println (text) // log.println (text)
R2: this.activate () -> stdout.println ("Activation") ; this.activate () ; stdout.println ("Activated")
```

Lorsque la méthode `activate` est invoquée sur l'interation *i*, aucune règle d'interaction de *i* n'est encore présente dans le système [1]. Ceci est en particulier vrai pour la règle d'interaction décrivant le comportement réactif associé au message déclencheur `this.activate ()`. Par conséquent, le comportement réactif (et donc son exécution) associé à la méthode `activate` sera purement et simplement ignoré.

Supposons maintenant que seule la règle d'interaction dont `this.activate ()` est le message déclencheur (règle `R2` ci-dessus) soit présente dans le système dès la création de l'interaction. Dans ce cas, l'invocation de la méthode `stdout.println (...)` réalisée avant celle du message déclencheur ne prend pas en compte le comportement réactif défini par la règle d'interaction `R1` tandis que celle réalisée après l'invocation du message déclencheur va le prendre en compte.

Le seul bémol à cette solution est que rien dans l'écriture de la règle d'interaction `R2` ne permet de se rendre compte de cette différence de sémantique entre les deux invocations à la méthode `stdout.println (...)`.

Inhibition de l'interaction

De manière similaire à l'étape d'activation, il est possible d'inhiber temporairement les comportements réactifs définis par une interaction. Il s'agit donc de l'opération inverse de l'activation. Cette étape est réalisée par invocation de la méthode `unactivate`. Cette méthode supprime, par invocation de la méthode `unregisterRule` définie par la classe `MetaObject`, toutes les règles d'interaction qui ont été ajoutées par la méthode `activate` aux métaobjets des objets participants (figure 8.12).

Là aussi, cette inhibition est réalisée de manière atomique vis-à-vis des objets participants et peut être personnalisée par la définition d'un comportement réactif sur la méthode `unactivate`.

8.4.3 Destruction d'une interaction

Une interaction, étant un objet, peut être détruite. Dans ce cas, elle n'influence plus les objets participants. Cependant la destruction définitive [2] d'une interaction revêt un caractère particulier lié, comme

1. Hormis, d'une certaine manière, la règle d'interaction `this.init ()` (se reporter au paragraphe 8.4.1).

2. On rappelle qu'une interaction peut être temporairement inhibée (cf. paragraphe ci-dessus).

```
procédure Scheme::unactivate ();
début
    pour chaque règle d'interaction r de l'interaction  faire début
        soit mo le métaobjet associé au message déclencheur de r;
        mo.unregisterRule (r)
    fin
fin
```

FIG. 8.12 – *Algorithme d'inhibition d'une interaction*

pour l'instanciation de l'interaction, aux informations supplémentaires nécessaires à l'interaction tout au long de sa vie. Le concept de destructeur du modèle à objets permet la modification de la sémantique de la destruction des instances d'une classe.

Ainsi, le destructeur d'une interaction, noté `uninitialize`, redéfini par la classe `Scheme` ou l'une de ses sous-classes, est responsable de la suppression de l'interaction dans le système (figure 8.13). Il invoque tout d'abord, si l'interaction est toujours activée dans le système au moment de sa destruction, la méthode `unactivate`. Ensuite, il invoque la méthode `finish`. Par défaut, cette méthode ne fait rien. Son but étant d'offrir un moyen pour exprimer, par un comportement réactif notamment, une action sur les objets participants juste avant la destruction de l'interaction.

```
procédure Scheme::uninitialize (Object o);
début
    si l'interaction est toujours activée  alors
        unactivate ()
    si il existe une règle d'interaction r dont le message déclencheur est this.finish  alors
        exécuter le comportement réactif de r
    sinon
        finish ()
    super::uninitialize (o)
fin
```

FIG. 8.13 – *Algorithme de destruction d'une interaction*

8.5 Dépôt des schémas d'interactions

Les modèles d'architectures distribuées proposant des mécanismes de découverte dynamique de nouveaux services (CORBA, Java RMI, *.Net* framework par exemples) offrent un mécanisme de dépot des services distribués (*service repository*) sous la forme de méta-informations. Par exemple en CORBA, le modèle *Object Management Architecture* (OMA) définit le concept de dépôt d'interfaces (*Interface Repository*) permettant de déposer dans une « base de données » un ensemble de méta-informations concernant les interfaces des objets distribués présents sur le bus CORBA. Ainsi une représentation objet du code IDL est stockée dans une base de données accessible à partir du bus réseau (l'ORB).

Ainsi, les schémas d'interactions étant des interfaces, ils disposent d'une représentation, disponible à l'exécution, dans ce dépôt d'interfaces. Cependant cette représentation est incomplète puisque les comportements réactifs, étant des données statiques (similaires aux variables de classes), ne peuvent être représentées en IDL. En fait le dépôt d'interfaces offre les mécanismes nécessaires pour parcourir le graphe des interfaces IDL, mais n'offre aucun support au parcours du graphe des comportements réactifs définis dans un schéma d'interactions. Pour ce faire, le modèle à interactions distribuées inclut un mécanisme de dépot de schémas d'interactions et rend ce dépot de schémas d'interactions accessible à distance via le bus réseau.

Ainsi, un dépôt de schémas d'interactions (*Interaction Scheme Repository*, ISR) est un service qui fournit une représentation objet des informations d'un code écrit en ISL concernant les schémas d'interactions sous la forme de « méta-descriptions » disponibles lors de l'exécution des applications. Les informations du dépôt de schémas d'interactions peuvent être utilisées par le modèle à interactions pour vérifier, par exemple, la validité d'une pose d'un schéma d'interactions sur un ensemble d'objets.

De plus, grâce aux informations contenues dans le dépôt de schémas d'interactions, une application peut découvrir un schéma d'interactions qui n'était pas connu lors de la compilation et être en mesure de déterminer quelles sont ses caractéristiques (et notamment le type des objets sur lesquels peut être posé le schéma d'interactions). Ainsi un dépôt de schémas d'interactions est une représentation, sous la forme de « méta-informations », de schémas d'interactions définis à l'aide d'un code source ISL. Afin

de répondre parfaitement au caractère dynamique des interactions, les informations contenues dans un dépôt de schémas d'interactions peuvent être modifiées à l'exécution. De plus, le pouvoir d'expression du dépôt de schémas d'interactions est le même que celui du langage ISL.

8.5.1 Rôle du dépôt de schémas d'interactions

Un dépot de schémas d'interactions fournit un ensemble de fonctions pour interroger, stocker ou détruire les méta-descriptions ISL. Il est en tous points similaire au dépôt d'interfaces (*Interface Repository*) de CORBA. Il permet la manipulation de toute information ISL à partir de n'importe quel site distant (un navigateur est présenté dans la partie IV de ce mémoire de thèse).

À l'instar des butineurs des environnements de programmation, on peut consulter, ajouter ou supprimer les schémas d'interactions du dépôt de schémas d'interactions. Une fois l'information stabilisée, le source ISL peut être reconstuit directement à partir des méta-informations contenues dans un dépôt de schémas d'interactions.

Les mécanismes du modèle à interactions peuvent utiliser les définitions des schémas d'interactions maintenues par un dépôt de schémas d'interactions pour (la figure 8.14 montre des utilisations possibles du dépot de schémas d'interactions) :

– Vérifier une hiérarchie de schémas d'interactions : le mécanisme du modèle à interactions en charge de l'ajout et de la suppression d'un schéma d'interactions dans le système peut vérifier la conformité du graphe d'héritage. Il peut ainsi vérifier que, lors de l'ajout, tous les schémas d'interactions dont dérive un schéma d'interactions sont bien présents dans le dépôt et que, lors de la suppression, aucun schéma d'interactions n'hérite du schéma d'interactions qui doit être supprimé.
– Vérifier les droits d'accès à un schéma d'interactions : le dépôt de schémas d'interactions peut vérifier que l'opération demandée par une application est autorisée (notion de sécurité).

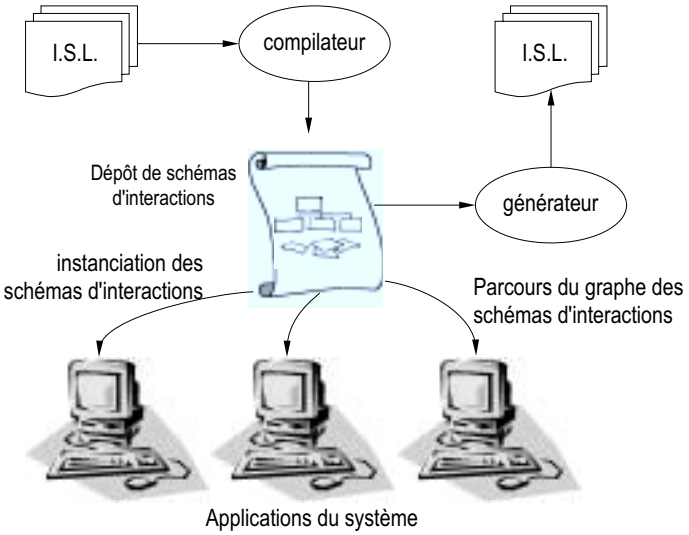


FIG. 8.14 – *Utilisations possibles du dépôt de schémas d'interactions*

Les définitions des schémas d'interactions maintenues par le dépôt de schémas d'interactions étant publiques, ces définitions peuvent aussi être utilisées par les clients et d'autres services. Par exemple, le dépôt de schémas d'interactions peut servir à :

– Administrer les schémas d'interactions :
  Le client peut gérer l'installation et la distribution des schémas d'interactions à travers le réseau.
– Construire des outils :
  Le dépôt de schémas d'interactions fournit des méta-informations permettant de construire des outils de style CASE comme, par exemple, des butineurs (*browsers*) de schémas d'interactions, des compilateurs ISL, etc.

Le dépôt de schémas d'interactions joue un rôle majeur dans cette architecture distribuée, tout comme le dépôt d'interfaces de CORBA pour l'ORB. En effet, c'est lui qui rend accessible aux ap-

plications les informations sur les schémas d'interactions *à l'exécution*, informations *indépendantes* de tout langage.

Des opérations sont définies permettant de manipuler le contenu du dépôt de schémas d'interactions. D'autres opérations permettent d'extraire un ensemble d'informations décrivant dans leur intégralité un schéma d'interactions ou une règle d'interaction.

Un dépôt de schémas d'interactions offre une notion de droit d'accès aux différentes opérations possibles sur les schémas d'interactions. En effet, un schéma d'interactions ne doit pas nécessairement pouvoir être détruit par n'importe quel utilisateur dans le système. De même, un butineur, qui surveillerait l'état du dépôt de schémas d'interactions par le biais d'interactions posées entre les objets du dépôt de schémas d'interactions et ceux du butineur, peut vouloir interdire aux autres applications de modifier ou supprimer les descriptions des schémas d'interactions et des interactions qu'il a définis.

Le modèle à interactions distribuées définit deux classes d'utilisation, chaque classe ayant des droits d'accès par défauts différents. Le choix de la classe d'utilisation par une application est réalisée lors de sa connexion au dépôt de schémas d'interactions.

Les droits d'accès en fonction de la classe d'utilisation sont les suivants :

- La classe *utilisateur* :
  L'application qui enregistre un schéma d'interactions sera la seule à pouvoir modifier ou supprimer les méta-informations du schéma d'interactions dans le dépôt de schémas d'interactions, mais toute application du système sera en mesure d'accéder en lecture au schéma d'interactions et de l'instancier. De plus, lorsque l'application cessera d'exister dans le système, tous les schémas d'interactions (et indirectement les interactions instances de ces schémas d'interactions) définis par cette application avec l'accès *utilisateur* seront automatiquement supprimés du dépôt de schémas d'interactions.
- La classe *administrateur* :
  L'application qui enregistre le schéma d'interactions va partager, aussi bien en lecture qu'en écriture, les méta-informations du schéma d'interactions avec toutes les applications du système. De plus, sauf pour le cas décrit ci-après, les méta-informations du schéma d'interactions ne seront pas automatiquement enlevées du dépôt de schémas d'interactions, ni toutes les interactions instances de ce schéma d'interactions détruites, lorsque l'application cessera d'exister dans le système.
  Il est à noter, cependant, qu'un schéma d'interactions définissant sa propre classe et enregistré avec les droits d'accès *administrateur* sera automatiquement supprimé du dépôt de schémas d'interactions lorsque l'application qui l'a défini cessera d'exister dans le système.

8.5.2 Nommage des schémas d'interactions dans un dépôt de schémas d'interactions

Les schémas d'interactions étant des classes, ils disposent d'un nom permettant de les désigner. Le modèle à objets impose que chaque classe (et par conséquent chaque schéma d'interactions) dispose d'un nom qui lui soit propre et qui soit différent des noms des autres classes du système. Par conséquent, le nom d'un schéma d'interactions permet d'identifier sans ambiguïté ce dernier. A priori, le dépôt de schémas d'interactions peut donc utiliser le nom des schémas d'interactions comme clef primaire pour permettre l'accès aux méta-informations des schémas d'interactions.

Cependant, un nom de classe est défini par rapport à un espace de nommage donné. Cet espace de nommage participe à la résolution du nom complet de la classe (qui doit lui aussi être unique). Ainsi, dans deux espaces de nommages différents, deux classes peuvent avoir le même nom « relatif ».

Par conséquent, le modèle à interactions distribuées doit prendre en compte ce concept d'espace de nommage pour éviter d'avoir deux schémas d'interactions possèdant le même nom (dans deux espaces de nommages différents). Ainsi, le modèle à interactions distribuées va représenter, dans le dépôt de schémas d'interactions, les différents espaces de nommages par la notion de dossier : une arborescence hiérarchique de dossier est donc mise en place dans le dépôt de schémas d'interactions.

8.6 Conclusion

Dans ce chapitre, nous avons présenté, de manière détaillée, une description structurelle et fonctionnelle de la projection de l'architecture du modèle à interactions distribuées dans un langage tel que C++ ou Java. Cette projection est basée sur la définition d'un noyau de sept classes (dont quatre métaclasses).

Nous avons présenté les relations entre les comportements de ces différentes classes par le biais d'algorithmes. Nous avons également détaillé le fonctionnement des comportements réactifs et notamment précisé qu'ils étaient les responsables de la fusion comportementale.

Cette description structurelle et fonctionnelle a été validée lors de la mise en œuvre du modèle à interactions distribuées à la fois en C++ / CORBA (que nous décrivons de manière détaillée dans les deux chapitres suivants) et en Java.

Chapitre 9

Mise en œuvre en C++ et CORBA *: utilisation de schémas de conception*

> « *The design patterns require neither unusual language features nor amazing programming tricks [...]. All can be implemented in standard object-oriented languages, though they might take a little more work than* ad hoc *solutions. But the extra effort invariably pays dividends in increased flexibility and reusability.* » [GHJ+95]

Dans ce chapitre, nous nous concentrons sur la mise en œuvre de la gestion de l'exécution des interactions dans le langage compilé et fortement typé qu'est C++ [Str91]. Ce chapitre complète donc la description de l'architecture du modèle à interactions distribuées définie dans le chapitre précédent.

L'exécution des comportements réactifs est basée sur une extension de la sémantique d'envoi de messages. Ceci implique, d'une part, une évaluation dynamique des messages (dont la problématique est présentée ci-après) et, d'autre part, un contrôle des messages (présenté au chapitre 7). Ainsi, dans ce chapitre, nous présentons notre solution pour évaluer de manière dynamique des messages dans l'environnement compilé et fortement typé qu'est C++. Ensuite nous décrivons notre solution pour la capture et le contrôle des messages puis une mise en œuvre possible du dépôt de schémas d'interactions. Finalement, nous présentons les limites de notre mise en œuvre vis-à-vis de l'architecture décrite dans le chapitre précédent.

9.1 Problématique : évaluation dynamique d'un message

Lors de l'envoi d'un message à un objet, deux situations peuvent se produire. Si aucun comportement réactif n'est associé à ce message alors, dans ce cas, le message est normalement résolu, et donc la méthode est cherchée puis appliquée normalement. Si au moins un comportement réactif est associé à ce message alors le message est contrôlé. Lorsqu'un message est contrôlé cela implique l'exécution d'une règle d'interaction (issue, si nécessaire, de la fonction de fusion comportementale) à la place du message originel.

Cette exécution implique l'exécution de chacun des opérateurs réactifs contenu dans la règle d'interaction. Si l'exécution de certains opérateurs ne pose aucun problème particulier, l'exécution des messages contenus dans la règle d'interaction (opérateur `msg`) est très délicate. En effet, cela implique l'invocation « physique » de la méthode désignée par l'opérateur réactif. Or celui-ci la désigne uniquement par son nom (une chaîne de caractères).

Or, nous nous plaçons dans un contexte compilé. Ainsi, il n'est pas possible d'évaluer simplement une méthode à partir de son nom comme cela est le cas dans un contexte interprété. Par conséquent, il est

nécessaire de mettre en œuvre un mécanisme d'évaluation dynamique des messages. Ce mécanisme doit être générique, c'est-à-dire qu'il doit être en mesure d'évaluer n'importe quelle méthode sur n'importe quel objet. En effet, il est impossible de savoir, à priori, quelles vont être les méthodes susceptibles de nécessiter une évaluation dynamique. Ceci est une conséquence de la propriété de dynamicité de la définition des schémas d'interactions.

La solution, décrite en détail dans les paragraphes suivants, consiste à définir une table de correspondances dont la clef est le nom (sa signature) de la méthode et le nom de la classe de l'objet, la valeur associée à cette clef étant une représentation réifiée de la méthode. En effet, il n'est possible d'invoquer une méthode en lui fournissant un nombre d'arguments connus uniquement lors de l'invocation (et non lors de la compilation).

Malgré tout, cette solution nécessite la mise en place de tout un environnement pour satisfaire aux conditions de typage imposées par le langage. En effet, lorsqu'un message est contrôlé par le mécanisme de capture, ses paramètres doivent notamment pouvoir être transmis au mécanisme gérant l'exécution des comportements réactifs (et donc au mécanisme d'évaluation dynamique des messages).

Or chaque message dispose de sa propre signature, et donc de ses propres types de paramètres, tandis que les mécanismes d'exécution des comportements réactifs et d'évaluation dynamique des messages sont des mécanismes « génériques » pouvant être appliqués à n'importe quel message. Il est donc nécessaire d'encapsuler (par réification) les paramètres des messages contrôlés afin de les transmettre aux mécanismes d'exécution des comportements réactifs et d'évaluation dynamique des messages.

Le mécanisme DII de CORBA répond à ce problème d'évaluation dynamique des messages mais uniquement pour des objets CORBA. Or, l'un de nos critères initiaux stipule qu'aucune discrimination ne doit être faites entre les objets. Par conséquent, les objets interagissants ne doivent pas être nécessairement des objets CORBA.

9.2 Notre solution pour l'évaluation dynamique des messages

Notre solution consiste à encapsuler chaque méthode publique d'un objet interagissant dans un objet permettant l'invocation de cette méthode depuis les autres objets du système. Afin de respecter le typage fort imposée par le langage cible, une réification des paramètres est également mise en place. C'est cette représentation des paramètres qui est passée à l'objet encapsulant la méthode.

9.2.1 Réification des méthodes

Cette solution permet à un objet d'exécuter la méthode encapsulée sans rien connaître à priori sur cette méthode. La mise en œuvre de cette solution est basée sur le schéma de conception Commande (*Command Design Pattern* [GHJ+95, pp. 233–242]) et présentée par la figure 9.1.

Ainsi, en réifiant la méthode, le mécanisme d'exécution des comportements réactifs des interactions peut invoquer l'exécution de cette méthode puisque cette invocation est elle-même un objet (l'objet encapsulant la méthode) : « *The Command pattern let objects make requests of unspecified application objects by turning the request itself into an object.* » [GHJ+95, p. 233].

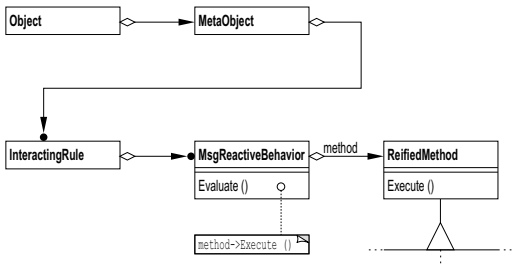


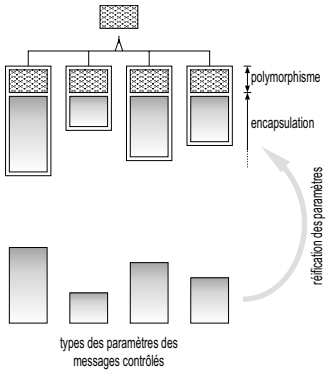
FIG. 9.1 – *Instanciation du schéma de conception Commande dans le cadre de la réification des méthodes*

En effet, la connaissance requise pour exécuter la méthode (et en particulier la façon de désemballer les paramètres réifiés) est entièrement contenue dans l'objet réifiant cette dernière. De ce fait, le mécanisme d'exécution du comportement réactif d'envoi de messages n'a pas besoin de disposer de la connaissance nécessaire à l'exécution de la méthode.

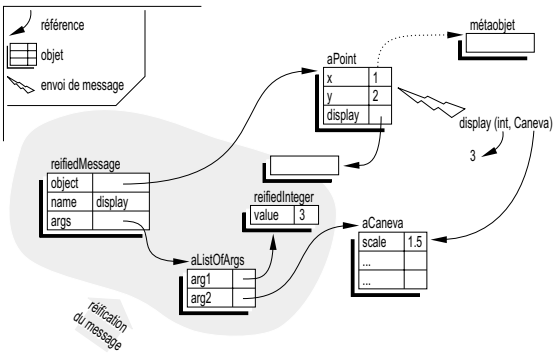
9.2.2 Réification des paramètres

Le mécanisme d'encapsulation que nous proposons est proche de celui défini dans la première version d'Open C++ [Chi93] et consiste en une réification des paramètres du message. En fait les paramètres du message contrôlé sont « empaquetés » (*packaged*) et regroupés dans un objet conteneur (une

liste). Ces paramètres peuvent ainsi être récupérés à partir de cette liste et « désempaquetés » (*unpackaged*) avant d'être utilisés. L'objet conteneur peut être considéré comme le bloc de pile de l'envoi du message.



9.2a – Mécanisme d'encapsulation des types des paramètres



9.2b – Réification des messages

FIG. 9.2 – *Réification des paramètres*

Notre mécanisme de réification des paramètres est basé sur les concepts d'encapsulation et de polymorphisme du modèle à objets sous-jacent. En effet, l'encapsulation va réaliser la réification proprement dites, tandis que le polymorphisme va rendre tous les paramètres réifiés compatibles vis-à-vis du typage sans toutefois perdre le typage d'origine des paramètres. Ce typage est en effet nécessaire lors de l'opération de déification [1].

Le mécanisme d'encapsulation est réalisé par application du schéma de conception (*design pattern*) nommé Stratégie (*Strategy Design Pattern* [GHJ+95, pp. 315–323]), tandis que le polymorphisme est basé sur le concept d'héritage de classes du modèle à objets (figure 9.2a). La figure 9.2b présente notre mécanisme de réification des messages. Elle montre notamment l'utilisation qui est faite du mécanisme d'encapsulation des paramètres et la manière dont les informations constituant le message sont réifiées.

Afin que le support des interactions soit transparent du point de vue du programmeur, notre mécanisme d'encapsulation des paramètres des messages contrôlés doit lui aussi être transparent vis-à-vis du programmeur. Ceci implique que l'architecture du support des interactions doive mettre en œuvre des mécanismes d'emballage et de désemballage pour chacun des types des paramètres des messages susceptibles d'être contrôlés (en effet un message devient contrôlé lorsqu'un comportement réactif lui est associé).

La définition de ces mécanismes a forcément lieu lors de la phase de compilation [Ber99] et nécessite l'utilisation d'un compilateur « ouvert » (pré-compilateur ou méta-compilateur). En effet, l'instanciation de la classe encapsulant un type donné connu uniquement lors de l'exécution requiert la génération du code décrivant l'emballage et le désemballage pour ce type (le code de la classe). Cependant, cette classe à instancier n'est réellement connue que lors de l'exécution. Ainsi, le mécanisme d'instanciation standard tel que défini dans le modèle à objets ne permet pas de réaliser l'instanciation d'une classe non connue à la compilation. Heureusement, le schéma de conception nommé Constructeur Virtuel (*Virtual Constructor Design Pattern* [GHJ+95, pp. 107–116]) permet de résoudre ce problème.

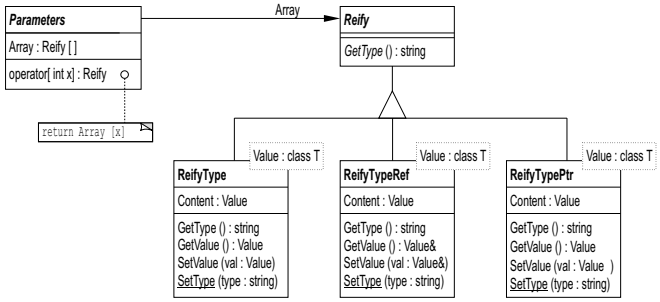


FIG. 9.3 – *Schéma UML des classes réifiant les paramètres*

1. La déification est l'opération inverse de la réification.

La figure 9.3 montre, sur un schéma de classes UML [BJR97], « l'instanciation » du schéma de conception Stratégie pour la réification des paramètres. On peut notamment voir que le mécanisme de réification est le même pour chaque style de types (« types simples », « types référencés » et « types pointés »). De ce fait nous avons utilisé le concept de classes génériques (*template classes*) et les avons paramétrées par le type réifié.

EXEMPLE. — CODE D'UNE MÉTHODE RÉIFIÉE

Nous présentons ci-dessous le code de la classe correspondante à l'entité`ReifiedMethod` de la figure 9.1 pour une méthode dont la signature est `aReturnType aClass::aMethod (aFirstType x, aSecondType * y)`.

On peut noter que l'on passe notamment comme paramètre à la méthode`execute` l'objet ainsi que la signature de la fonction ayant déclenchée l'interaction impliquant cette évaluation dynamique d'un message. En effet, si le message à exécuter est le message déclenchant alors il faut invoquer la méthode non contrôlée (son nom est suffixée avec`_original`). Dans tous les autres cas c'est la méthode contrôlée qui doit être est invoquée.

```
 1: class aClass_aMethod_aFirstType_aSecondTypePtr : public __FunctionCall
 2: {
 3:   public:
 4:     void execute (Reify *anObject, Parameters &params, Reify **ret, const char *pFctName, Reify *pObject);
 5:     {
 6:       aReturnType vResult;
 7:       aClass * pObjectCall = ((ReifyPtr <aClass> *)anObject)->GetValue ();
 8:
 9:       if (pObjectCall == ((ReifyPtr <aClass> *)pObject)->GetValue () && !strcmp (pFctName, "aMethod_aFirstType_aSecondTypePtr"))
10:         vResult = pOjectCall->aMethod_original (((ReifyType <aFirstType> *)(params [0]))->GetValue (),
11:                                                 ((ReifyTypePtr <aSecondType> *)(params [1]))->GetValue ());
12:       else
13:         vResult = pObjectCall->aMethod (((ReifyType <aFirstType> *)(params [0]))->GetValue (),
14:                                         ((ReifyTypePtr <aSecondType> *)(params [1]))->GetValue ());
15:
16:       if (ret)
17:         *ret = new ReifyType <aReturnType> (vResult);
18:       else if (params->GetSize () == 2)
19:         ((ReifyType <aReturnType> *)(params [2]))->SetValue (vResult);
20:     }
21: };
```

9.2.3 Détermination de la méthode réifiée

Réifier toutes les méthodes publiques de tous les objets interagissants n'est pas suffisant pour pouvoir évaluer dynamiquement cette méthode. En effet, encore faut il pouvoir déterminer l'objet encapsulant la bonne méthode. Puisque la classe à instancier qui réifie la bonne méthode est spécifique à une règle d'interaction, le mécanisme d'exécution des comportements réactifs ne peut prédire à la compilation quelle classe précise doit être instanciée — le mécanisme sait juste *quand* instancier la classe réifiant une méthode, mais pas *quelle* classe doit être instanciée.

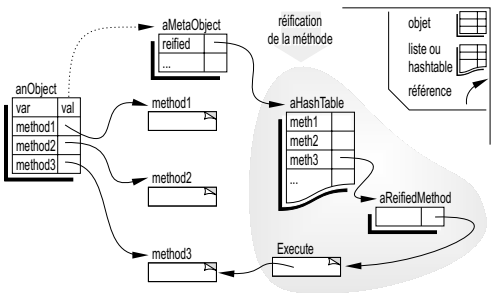


FIG. 9.4 – *Enregistrement des instances des méthodes réifiées*

Il est donc nécessaire d'extraire du mécanisme d'exécution des comportements réactifs la manière d'instancier la classe et de placer cette information ailleurs. Ceci peut être réalisé grâce à l'utilisation du schéma de conception de Constructeur Virtuel [GHJ+95, pp. 107–116].

L'utilisation de ce mécanisme d'instanciation par « constructeur virtuel » implique également d'enregistrer une instance de chacune de ces classes avant l'utilisation du mécanisme d'évaluation dynamique des messages afin de pouvoir associer au nom de la méthode à exécuter l'objet la réifiant (figure 9.4). Cet enregistrement peut être basée sur l'utilisation du schéma de conception Poids Mouche (Flyweight Design Pattern [GHJ+95, pp. 195–206]).

9.2.4 Mise en place de la solution

Afin que le support des interactions soit transparent du point de vue du programmeur, la réification des méthodes doit être entièrement à la charge de notre architecture. Or cette réification des méthodes implique la génération du code décrivant, pour chacune des méthodes à réifier, l'instanciation du schéma de conception Commande (cf. l'exemple montrant le code d'une méthode réifiée).

En effet, ce code, écrit dans le langage applicatif, ne peut pas être générique puisque seul l'objet « Commande » dispose de la connaissance requise pour exécuter la méthode et effectuer les vérifications de typage. La génération de ce code a nécessairement lieu lors de la phase de compilation. De plus, elle nécessite l'utilisation d'un compilateur « ouvert » (ce compilateur pouvant être le même que celui utilisé pour la réification des types des paramètres). Dans notre cas, ce compilateur est Open C++ v2 [Chi95].

Il est à noter que le mécanisme d'évaluation dynamique des messages fait partie intégrante de la métaclasse `MetaObject` définie dans le chapitre précédent (puisque c'est elle qui a la charge de l'exécution des comportements réactifs). On peut remarquer que cette métaclasse n'existe pas en tant que tel lors de l'exécution du programme.

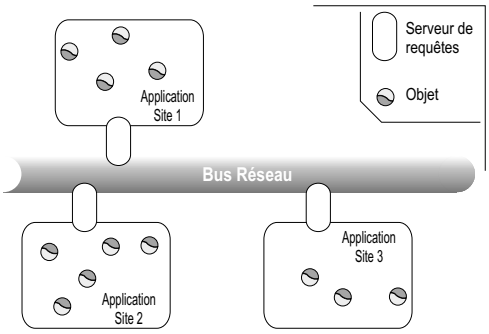
Sa mise en œuvre se compose, en effet, d'un ensemble de classes dont certaines sont générées lors de la compilation (réification des méthodes et des types des paramètres). En fait, nous utilisons le MOP à la compilation d'Open C++ v2 pour définir notre propre MOP à l'exécution (ce dernier étant spécialisé dans le support des interactions).

9.2.5 Extension de la solution aux évaluations dynamiques de messages distants

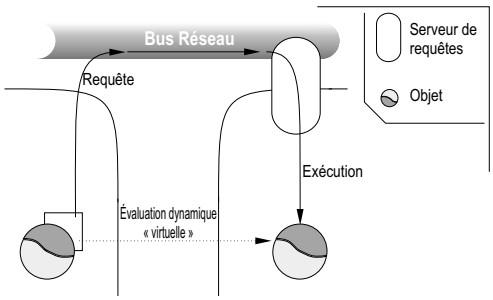
Il peut être nécessaire d'évaluer dynamiquement un message dont l'objet receveur (le destinataire) est situé sur un site distant de celui de l'objet émetteur (celui qui a déclenché l'interaction).

Si l'objet destinataire n'est pas un objet distribué, alors l'évaluation dynamique du message doit être réalisée sur le site de l'objet destinataire. Ceci implique de déléguer l'évaluation du message au mécanisme d'évaluation dynamique des messages du site contenant l'objet destinataire. Ainsi, le mécanisme d'évaluation dynamique des messages doit être accessible à distance (sur le bus logiciel).

Pour ce faire, nous définissons un serveur de requêtes dont le rôle est de traiter les demandes d'évaluation dynamique des messages provenant des autres sites (figure 9.5). L'exécution des messages transmis au serveur de requêtes fait appel au mécanisme d'évaluation dynamique des messages présent en local sur le site. Ainsi, les requêtes contiennent, d'une part, l'ensemble des informations nécessaires pour déterminer sans ambiguïté la méthode à exécuter que et, d'autre part, la liste des arguments à transmettre lors de l'appel de la méthode. Une fois ces éléments obtenus par le serveur de requêtes, le message est exécuté comme cela a été décrit ci-dessus.



9.5a – Vue d'ensemble du serveur de requêtes



9.5b – Cheminement d'une requête

FIG. 9.5 – *Le serveur de requêtes*

Si l'objet destinataire est un objet distribué (un objet CORBA dans notre cas) le mécanisme d'invocation dynamique (DII) peut être utilisé pour réaliser cette évaluation du message. Ainsi, le site où se trouve l'objet émetteur du message (celui qui a déclenché l'interaction) construit une requête DII CORBA et l'envoit au bus logiciel. Par conséquent, cette solution évite d'avoir recours au serveur de requêtes que nous venons de définir et, de ce fait, à réifier l'exécution du message aussi bien du côté émetteur que du côté récepteur.

Il est à noter qu'aussi bien le serveur de requêtes que le mécanisme DII contribuent activement au fait que l'exécution des interactions est entièrement transparente du point du programmeur.

9.3 Notre solution pour la capture des messages

Offrir un mécanisme permettant l'évaluation dynamique des messages est certes nécessaire mais pas suffisant pour permettre l'exécution des interactions. En effet, il faut pouvoir déclencher ce mécanisme lorsqu'un objet reçoit un message. C'est le but du mécanisme de capture des messages.

Notre mécanisme de contrôle et de capture de l'envoi de message est très similaire à celui d'Open C++ v1.0 présenté dans le paragraphe 7.2.4. Cependant, afin d'éviter le sur-coût induit par la réflexivité à l'exécution, la mise en place du contrôle a lieu durant la phase de compilation et non lors de l'exécution. Ceci implique que le mécanisme de « détournement » des envois de messages vers le métaobjet est réalisée lors de la phase de compilation.

9.3.1 Mécanisme du *wrapper*

Ce détournement est basé sur le mécanisme dit du *wrapper* [Bra96, BFJ+]. Il consiste, pour chaque méthode contrôlée (il s'agit de toutes les méthodes d'interface), à la renommer et à la redéfinir avec le code du contrôle de la méthode (c'est par conséquent cette nouvelle méthode qui sera invoquée lors d'une invocation destinée à la méthode originelle). La figure 9.6 présente ce mécanisme.

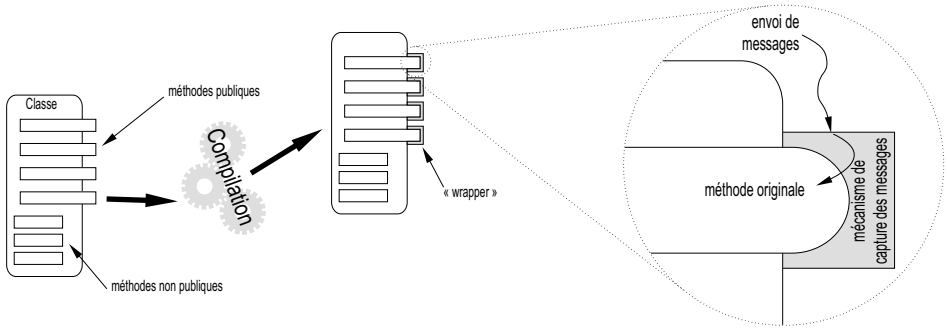


FIG. 9.6 – *Principe du mécanisme de capture de l'envoi de messages*

La figure 9.7 présente notre mise en œuvre du mécanisme dit du « wrapper ». Elle précise le fait que la méthode originelle (il s'agit de la méthode qui est capturée) n'est pas stockée dans le dictionnaire des méthodes mais que la référence à la méthode originelle dans le dictionnaire est remplacée par celle désignant la méthode de substitution. C'est cette méthode de substitution qui dispose d'une référence à la méthode originelle.

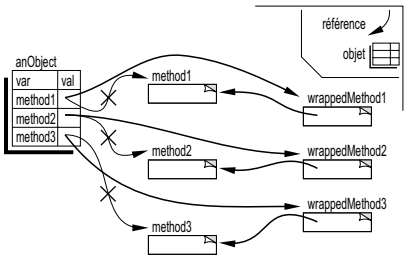


FIG. 9.7 – *Mise en œuvre de la capture des messages*

Ainsi, le contrôle des messages reste présent à l'exécution. Ceci est en effet indispensable puisqu'il évolue de manière parallèle à la définition dynamique des interactions. Le métaobjet est donc composé de deux parties disjointes : l'une, existante à la compilation, et mettant en place le contrôle des messages, l'autre, existante à l'exécution, et exécutant la sémantique du contrôle des messages.

9.3.2 Notre mise en œuvre

Le code ci-dessous (en C++) fournit une description de cette prise de contrôle et permet de préciser les choix effectués (le code en italique décrit la partie de la sémantique du contrôle associée à la capture des messages et présentée plus loin dans ce paragraphe).

```
1:  returnType aClass::aControledMethod (args)
2:  {
3:    if (hasInteraction () && hasReactiveBehavior ("aControledMethod"))
4:      metaobject.control (reified_args);
5:    else
6:      initial_aControledMethod (args);
7:  };
```

Parmi ces choix, l'on peut noter que le contrôle du message n'est pas systématiquement réalisé mais est subordonné à la présence d'une interaction et plus précisément d'un comportement réactif associé au message (ligne 3). De ce fait, le contrôle effectif du message n'est réalisé que lorsqu'il est nécessaire, ce qui permet de limiter le sur-coût lié à la réification des paramètres du message (ligne 4). Lorsque le message n'a pas à être contrôlé il est transmis tel quel à la méthode originelle (ligne 6). Il est à noter que la ligne 4 représente la description de la sémantique du contrôle par appel du métaobjet.

La mise en place de ce mécanisme de capture implique l'utilisation d'un compilateur « ouvert » (réflexif) disposant d'un métaobjet présent lors de la phase de compilation (phase durant laquelle le mécanisme est intégré à l'application). En effet, les langages cibles ne disposent d'aucun mécanisme réflexif. Dans notre cas, ce compilateur est Open C++ v2 [Chi95]. Il est à noter que ce mécanisme de capture des messages fait partie intégrante de la métaclasse `MetaObject` définie dans le chapitre précédent. Or ce mécanisme, bien que présent à l'exécution, est mis en place lors de la phase de compilation. Ceci signifie que la partie de la métaclasse `MetaObject` en charge de ce mécanisme n'existe qu'à la compilation.

EXEMPLE. — CODE D'UNE MÉTHODE CAPTURÉE

Nous présentons ci-dessous le code du mécanisme de capture pour une méthode dont la signature est la suivante : `aReturnType aClass::aMethod (aFirstType x, aSecondType * y)`.

On peut noter qu'il est fait appel à la classe `interactionSystem` pour obtenir la liste des règles d'interaction définies sur la méthode contrôlée. De plus, dans un soucis d'optimisation, les objets réifiant les paramètres de la méthode sont obtenus à partir d'un « fond commun d'objets » (*object pool*), ce qui évite des création et destruction d'objets intempestives.

```
 1: class aClass
 2: {
 3:   public:
 4:     aReturnType aMethod (aFirstType x, aSecondType * y);
 5:
 6:     aReturnType aMethod_original (aFirstType x, aSecondType * y)
 7:     {
 8:       code initial de class::aMethod
 9:     }
10: };
11:
12: aReturnType aClass::aMethod (aFirstType x, aSecondType * y)
13: {
14:   Array < InteractingRule > * reactiveBehavior;
15:   if (interactionSystem::hasInteraction (this) &&
16:       (reactiveBehavior = interactionSystem::hasReactiveBehavior (this, "aMethod_aFirstType_aSecondTypePtr")))
17:   {
18:     Array < Reify * > parameters;
19:     ReifyType < aReturnType > * reified_result = interactionSystem::getPoolType ("aReturnType");
20:     ReifyType < aFirstType > * reified_x = interactionSystem::getPoolType ("aFirstType");
21:     ReifyTypePtr < aSecondType > * reified_y = interactionSystem::getPoolTypePtr ("aSecondType");
22:     aReturnType result;
23:
24:     reified_x->SetValue (x);
25:     parameters->add (reified_x);
26:
27:     reified_y->SetValue (y);
28:     parameters->add (reified_y);
29:     parameters->add (reified_result);
30:
31:     interactionSystem::executeReactive (this, "aMethod_aFirstType_aSecondTypePtr", parameters, reactiveBehavior);
32:     result = reified_result->GetValue ();
33:
34:     interactionSystem::setPoolType ("aReturnType", reified_result);
35:     interactionSystem::setPoolType ("aFirstType", reified_x);
36:     interactionSystem::setPoolTypePtr ("aSecondType", reified_y);
37:
38:     return result;
39:   }
40:   else
41:     return aMethod_original (x, y);
42: }
```

9.4 Notre solution pour la mise en œuvre du dépôt de schémas d'interactions

Nous décrirons dans ce paragraphe une mise en œuvre possible du dépôt de schémas d'interactions sous la forme d'un service CORBA en définissant un ensemble d'interfaces en IDL [Lam87] permettant d'accéder aux méta-informations contenues dans un dépôt de schémas d'interactions.

9.4.1 Architecture du dépôt de schémas d'interactions

Comme cela a été vu dans le chapitre 5, les schémas d'interactions doivent être enregistrés sous la forme de méta-informations dans un dépôt de schémas d'interactions. C'est par le biais de ces méta-informations, disponibles lors de l'exécution des applications, qu'il sera possible de créer des interactions, instances des schémas d'interactions.

Nous allons représenter les méta-informations du dépôt de schémas d'interactions sous la forme d'objets accessibles au travers d'un ensemble d'interfaces CORBA définies en IDL. Ainsi chaque « type » de méta-informations sera représenté par une ou plusieurs interfaces IDL.

Le dépôt de schémas d'interactions utilise une structure hiérarchique arborescente permettant de grouper des schémas d'interactions sous un nom commun, dans un *dossier*. Cette structure hiérarchique à dossiers est très similaire à celle définie par le service CORBA de nommage (CORBA *Naming Service*) [OMG00]. Ainsi, comme pour le service de nommage, un schéma d'interactions est référencé par une séquence de dossiers, cette séquence définissant le chemin du schéma d'interactions au sein de la structure hiérarchique arborescente. Ce chemin définit l'espace de nommage du schéma d'interactions et est utilisé pour nommer complètement le schéma d'interactions (nom absolu). De plus, la représentation des méta-informations est fortement inspirée de celle proposée par l'OMG pour le dépôt d'interfaces CORBA (*Interface Repository* [OMG01, chap. 10])

Chaque feuille de cette structure arborescente hiérarchique de dossiers est un ensemble de méta-informations qui représentent un schéma d'interactions. Ces méta-informations sont elles-mêmes structurées sous la forme d'une arborescence hiérarchique afin de respecter l'arbre de syntaxe abstraite du schéma d'interactions qu'elles représentent. Il est à noter que ces méta-informations sont une projection en IDL des métaobjets et des classes des opérateurs réactifs présentés au chapitre 8.

Types des objets de dépôt

Les méta-informations contenues par un dépôt de schémas d'interactions (*Interacting Scheme Repository*, ISR) sont définies sous la forme d'une collection d'*objets de dépôt* suivants :

- **`Repository`** : la racine de l'espace de nommage du dépôt de schémas d'interactions. Il contient des définitions de dossiers.
- **`Folder`** : une définition d'un dossier. Il permet de regrouper un ensemble de schémas d'interactions. Il contient des définitions de schémas d'interactions et d'autres dossiers.
- **`InteractionScheme`** : une définition d'un schéma d'interactions. Il contient la définition des règles d'interaction définissant les comportements interagissants du schéma d'interactions.
- **`InteractingRule`** : une définition d'une règle d'interaction. Il contient un ensemble de variables et un comportement réactif (défini sous la forme d'un opérateur réactif).
- **`Operation`** : une description d'un comportement réactif. Chaque comportement réactif défini par le modèle à interactions distribuées dispose de sa propre interface d'objet de dépôt basée sur l'interface `Operation`.
- **`Variable`** : une description d'une variable (son type) utilisée dans le comportement réactif d'une règle d'interaction.

Pour chacun des objets de dépôt, la spécification de l'interface CORBA liste les attributs définis par cet objet. La plupart de ces attributs correspondent directement aux déclarations du langage ISL.

La relation de contenance pour les types d'objets de dépôt dans le dépôt de schémas d'interactions est présentée par la figure 9.8. Elle indique quels objets de dépôt sont contenus dans quels autres. Elle montre que chaque type d'objet de dépôt est soit un conteneur d'objets de dépôt, soit contenu par un autre objet de dépôt, soit les deux en même temps. Ainsi, afin d'uniformiser les fonctions de navigation entre les conteneurs et leur contenu, tous les objets qui contiennent d'autres objets de dépôt vont inclure un ensemble de méthodes permettant d'obtenir la liste des objets de dépôt contenus par le conteneur.

`Repository`	Chaque dépôt de schémas d'interactions est représenté par un objet de dépôt « racine ».
`Folder`	
`Folder` `InteractionScheme`	Un dossier décrit les sous-dossiers et les schémas d'interactions définis dans la portée (*scope*) du dossier.

`InteractionScheme`
`Variable`
`InteractingRule`
	Un schéma d'interactions décrit les règles d'interaction définies ou héritées par le schéma d'interactions et spécifie les schémas d'interactions dont il hérite.

`Variable` `Operation`	Une règle d'interaction représente un comportement réactif décrit à l'aide d'opérateurs réactifs sous la forme d'un arbre de syntaxe abstraite.

FIG. 9.8 – *Contenance des objets de dépôt*

Il y a au moins trois façons de localiser les méta-informations associées à un schéma d'interactions dans le dépôt de schémas d'interactions :

1. Obtenir un objet `InteractionScheme` à partir d'une interaction : Obtenir les méta-informations associées à un schéma d'interactions à partir d'une interaction est utile lorsque le schéma d'interactions définissant l'interaction n'est pas connu. En utilisant la méthode `getScheme` sur l'interaction, il est possible de récupérer une référence sur un objet `InteractionScheme` contenant, dans un dépôt de schémas d'interactions, les méta-informations du schéma d'interactions.
2. Obtenir un objet `InteractionScheme` à partir d'un autre schéma d'interactions : Obtenir les méta-informations associées à un schéma d'interactions *s* à partir des méta-informations d'un autre schéma d'interactions est utile lorsque, par exemple, les méta-informations d'un schéma d'interactions dont hérite le schéma d'interactions *s* sont recherchées. Il est possible, par exemple, d'obtenir la liste des méta-informations de tous les schémas d'interactions dont hérite un schéma d'interactions.
3. Naviguer à travers l'espace de nommage des dossiers : Naviguer à travers l'espace de nommage défini par les dossiers est utile lorsque les méta-informations sur une définition d'un schéma d'interactions particulier sont recherchées. En commençant la navigation à la racine du dépôt, il est possible d'obtenir toutes les définitions des schémas d'interactions enregistrés dans le dépôt de schémas d'interactions.

Définitions des types supportés

Plusieurs types sont utilisés par les interfaces CORBA des objets de dépôt (figure 9.9).

```
 1:  module CosInteraction
 2:  {
 3:    typedef string Identifier;
 4:    typedef string ScopedName;
 5:
 6:    enum ObjectKind
 7:    {
 8:      isrAll,
 9:      isrRepository,
10:      isrFolder,
11:      isrInteractionScheme,
12:      isrInteractingRule,
13:      isrVariable,
15:      isrOperation
16:    };
17:  };
```

FIG. 9.9 – *Types supportés par le dépôt de schémas d'interactions*

Les identificateurs (représentés par le type `Identifier`) sont des noms simples qui identifient les dossiers, les schémas d'interactions, les règles d'interaction, les variables et les opérations réactives. Ils correspondent exactement aux identificateurs du langage ISL. Un identificateur n'est pas nécessairement unique dans un dépôt de schéma d'interactions entier ; il est unique seulement dans un dépôt de

```
 1:  module CosInteraction
 2:  {
 3:    enum BadOperationReason
 4:    {
 5:      notRemoveable, hasDependency
 6:    };
 7:
 8:    exception BAD_OPERATION
 9:    {
10:      BadOperationReason reason;
11:    };
12:
13:    interface ISRObject
14:    {
15:      // read interface
16:      readonly attribute ObjectKind kind;
17:
18:      // write interface
19:      void destroy () raises (BAD_OPERATION);
20:    };
21:  };
```

FIG. 9.10 – *Interface commune à tous les objets de dépôt*

schéma d'interactions (représenté par un objet `Repository`), un dossier (représenté par un objet `Folder`), une définition d'un schéma d'interactions (représentée par un objet `InteractionScheme`), une règle d'interaction (représentée par un objet `InteractingRule`), ou une opération réactive (représentée par un objet `Operation`).

Un `ScopedName` est un nom composé d'un ou de plusieurs identificateurs séparés par les caractères "::". Un tel nom correspond à un nom qualifié du langage ISL. Un nom qualifié est dit *absolu* s'il commence par les caractères "::". Il décrit de manière non ambigu un objet de dépôt dans un dépôt de schéma d'interactions. Un nom qualifié est dit *relatif* s'il ne commence pas par les caractères "::". Son nom complet doit être résolu relativement à un contexte.

Un `ObjectKind` identifie le type réel d'un objet de dépôt et donc la méta-information contenue par cet objet de dépôt.

9.4.2 Interfaces de base

Plusieurs interfaces CORBA sont utilisées comme interfaces de base pour les objets de dépôt de l'ISR. Ces interfaces ne sont pas instanciables.

Un ensemble commun de fonctions est utilisé pour localiser les objets de dépôt dans le dépôt de schémas d'interactions (et donc les méta-informations). Ces fonctions sont définies dans les interfaces `ISRObject`, `Container`, et `Contained` décrites ci-après. Tous les objets de dépôt héritent de l'interface `ISRObject` qui fournit un moyen d'identifier le type réel de l'objet (et donc le type réel de la méta-information qu'il contient). Les objets de dépôt qui sont des conteneurs d'autres objets de dépôt (tels que les dossiers) héritent des fonctions de navigation de l'interface `Container`. Les objets contenus dans ces conteneurs héritent des fonctions de navigation de l'interface `Contained`.

Interface commune à tous les objets de dépôt

L'interface de base `ISRObject` représente l'interface la plus générique supportée par tous les objets de dépôt (figure 9.10). Toutes les autres interfaces sont dérivées de `ISRObject`.

L'attribut `kind` identifie le type exact de la méta-information (c'est-à-dire le type de l'objet de dépôt).

La fonction `destroy` détruit l'objet de dépôt (il cesse d'exister) et les méta-informations qu'il possède. Si cet objet est un conteneur (c'est-à-dire s'il dérive de l'interface `Container`) alors la fonction `destroy` est appliquée sur tout le contenu. Si l'objet de dépôt qui doit être détruit se trouve contenu dans un conteneur alors il est supprimé de ce conteneur.

Si la fonction `destroy` est invoquée sur un objet de dépôt `Repository` (c'est-à-dire que l'on essaye de détruire la racine du dépôt de schémas d'interactions) alors l'exception `BAD_OPERATION` est déclenchée avec `notRemoveable` comme raison. Si la suppression d'un objet de dépôt provoque une incohérence au sein du

dépôt de schémas d'interactions (par exemple la suppression d'un schéma d'interactions dont d'autres schémas d'interactions dérivent) alors l'exception BAD_OPERATION est déclenchée avec hasDependency comme raison.

Interface commune à tous les « contenants »

L'interface de base Contained est héritée par toutes les interfaces des objets de dépôt qui sont contenus dans d'autres objets de dépôt. Tous les objets du dépôt de schémas d'interactions excepté l'objet racine (Repository) et les définitions des comportements réactifs du langage ISL sont contenus dans d'autres objets de dépôts. L'interface Contained est présentée par la figure 9.11.

Un objet contenu dans un autre objet dispose d'un attribut name qui permet de l'identifier de manière unique dans le conteneur. Modifier l'attribut name permet de changer l'identité de l'objet de dépôt dans son conteneur. Une exception système BAD_PARAM est déclenchée, avec 3 comme code d'erreur (*minor code*), si un objet avec la même valeur pour l'attribut name existe déjà dans l'objet de dépôt conteneur.

Il dispose également d'un attribut owner qui permet d'identifier le propriétaire de la méta-information contenue dans l'objet de dépôt. Modifier l'attribut owner permet de changer le propriétaire d'une méta-information (par exemple pour rendre partagée la définition d'un schéma d'interactions). Une exception système BAD_PARAM est déclenchée, avec 2 comme code d'erreur, si la nouvelle valeur de l'attribut owner ne correspond à aucun propriétaire déclaré auprès du dépôt de schémas d'interactions.

```
 1:  module CosInteraction
 2:  {
 3:    typedef string Login;
 3:    typedef sequence <Container> ContainerSeq;
 4:
 5:    enum CannotProceedReason
 6:    {
 7:      nameAlreadyExist, badType, badLogin, notRemoveable
 8:    };
 9:
10:    exception CANNOT_PROCEED
11:    {
12:      CannotProceedReason reason;
13:    };
14:
15:    interface Contained : ISRObject
16:    {
17:      // read/write interface
18:      attribute Identifier          name;
19:      attribute Login               owner;
20:
21:      // read interface
22:      readonly attribute Container   definedIn;
23:      readonly attribute ScopedName  absoluteName;
24:
25:      struct Description
26:      {
27:        ObjectKind kind;
28:        any        value;
29:      };
30:
31:      Description describe ();
32:      ContainerSeq within ();
33:
34:      // write interface
35:      void move (in Container newContainer, in Identifier newName) raises (CANNOT_PROCEED);
36:    };
37:  };
```

FIG. 9.11 – *Interface commune à tous les « contenants »*

Les objets Contained disposent d'un attribut definedIn qui identifie le conteneur dans lequel l'objet est contenu. Les objets peuvent être contenus par un conteneur soit parce qu'ils sont définis avec ce dernier (par exemple, les règles d'interaction définies dans un schéma d'interactions sont contenues dans l'objet de dépôt décrivant les méta-informations du schéma d'interactions), soit parce qu'ils sont hérités par le conteneur (des règles d'interaction peuvent être contenues par un objet de dépôt décrivant un schéma d'interactions parce que le schéma d'interactions hérite d'un autre schéma d'interactions qui définit ces règles d'interaction). Si un objet est contenu dans un conteneur par héritage, l'attribut definedIn identifie l'objet de dépôt dont il est hérité.

L'attribut absoluteName est un nom qualifié absolu qui identifie l'objet contenu de manière unique dans le dépôt de schémas d'interactions représenté par un objet Repository. Si l'attribut definedIn de cet objet désigne l'objet de dépôt Repository, la valeur de l'attribut absoluteName est la concaténation de la chaîne "::" et du contenu de son attribut name. Dans les autres cas, la valeur de l'attribut absoluteName est la concaténation de la valeur de l'attribut absoluteName du conteneur référencé par l'attribut definedIn, de la chaîne de caractères "::", et du contenu de l'attribut name.

La fonction describe renvoit une structure contenant les méta-informations stockées par l'objet de dépôt. La description de la structure exacte associée à chaque type d'objet de dépôt est fournie lors de la définition de l'interface IDL du type de l'objet de dépôt. Le champ kind de la structure Description retournée fourni le type de l'objet le plus approprié vis-à-vis des énumérations définies dans ObjectKind et le champ value contient, sous la forme d'un objet any, les méta-informations décrites par le type de l'objet de dépôt.

La fonction within renvoie la liste des objets conteneurs dans lesquels est contenu l'objet de dépôt. Si l'objet de dépôt désigne un dossier ou un schéma d'interactions alors il ne peut être contenu que par un unique conteneur qui est nécessairement un objet de dépôt désignant un dossier (ou l'objet de dépôt Repository représentant le dépôt de schémas d'interactions lui-même). Les autres objets de dépôt peuvent être contenus par les objets qui les définissent et par ceux qui en héritent (par exemple un objet de dépôt décrivant les méta-informations d'une règle d'interation est contenu par l'objet de dépôt décrivant le schéma d'interactions qui l'a définie mais aussi par tous les objets de dépôt décrivant les méta-informations des schémas d'interactions héritant de ce schéma d'interactions).

La fonction move déplace l'objet de dépôt de son conteneur actuel vers un autre objet de dépôt conteneur. Elle enlève automatiquement l'objet de dépôt de son conteneur actuel, et l'ajoute dans le conteneur spécifié par newContainer. Afin que la cohérence du dépôt de schémas d'interactions soit toujours vérifiée après le déplacement de l'objet de dépôt, elle doit respecter les conditions suivantes :

- Le nouveau conteneur doit être en mesure de contenir le type de l'objet de dépôt. Si ce n'est pas le cas l'exception CANNOT_PROCEED est déclenchée avec badType comme raison.
- Le nouveau conteneur ne doit pas déjà contenir un objet de dépôt ayant la même valeur pour l'attribut name. Si cette condition n'est pas respectée l'exception CANNOT_PROCEED est déclenchée avec nameAlreadyExist comme raison.
- Si l'objet de dépôt ne peut pas être supprimé de son conteneur actuel (car sa suppression provoquerait une incohérence du dépôt de schémas d'interactions) alors l'exception CANNOT_PROCEED est déclenchée avec notRemoveable comme raison.

L'attribut name est modifié en newName. Les attributs definedIn et absoluteName sont mis à jour pour refléter le nouveau conteneur et le nom. Si l'objet de dépôt est aussi un conteneur alors l'attribut absoluteName de tous les objets qu'il contient sont également mis à jour.

Interface commune à tous les conteneurs

L'interface de base Container est utilisée pour mettre en œuvre une hiérarchisation des contenus tout en offrant une interface commune de navigation au sein des conteneurs. Un conteneur peut contenir un nombre quelconque d'objets de dépôt dérivés de l'interface Contained. Tous les conteneurs, excepté le conteneur Repository, sont aussi dérivés de l'interface Contained (ils sont tous contenus dans un autre conteneur). L'interface Container est présentée par la figure 9.12.

La fonction lookup renvoie un objet de dépôt contenu dans le conteneur à partir de son nom qualifié (relativement au conteneur). L'utilisation d'un nom qualifié absolu (c'est-à-dire débutant par "::") permet de renvoyer un objet de dépôt à partir de la racine du dépôt de schémas d'interactions. Si aucun objet de dépôt n'est trouvé, une référence sur l'objet *nil* est renvoyée.

```
 1:  module CosInteraction
 2:  {
 3:    typedef sequence <Contained> ContainedSeq;
 4:
 5:    interface Container : ISRObject
 6:    {
 7:      // read interface
 8:      Contained lookup (in ScopedName searchName);
 9:      ContainedSeq lookupName (in Identifier searchName, in long levels2Search, in ObjectKind limitKind,
                                  in boolean excludeInherited);
10:
11:      ContainedSeq contents (in ObjectKind limitKind, in boolean excludeInherited);
12:
13:      struct Description
14:      {
15:        Contained  containedObject;
16:        ObjectKind kind;
17:        any        value;
18:      };
19:
20:      typedef sequence <Description> DescriptionSeq;
21:
22:      DescriptionSeq describeContents (in ObjectKind limitKind, in boolean excludeInherited,
                                          in long maxItems);
23:    };
24:  };
```

FIG. 9.12 – *Interface commune à tous les conteneurs*

La fonction `contents` renvoie la liste des objets contenus ou hérités par l'objet de dépôt conteneur. Cette fonction permet de naviguer à travers la hiérarchie des objets de dépôt. En commençant la recherche à l'objet de dépôt `Repository`, un client peut utiliser cette fonction pour lister tous les objets de dépôt (et donc toutes les méta-informations) contenus dans le dépôt de schémas d'interactions.

Lorsque le paramètre `limitKind` a la valeur `isrAll`, tous les objets du conteneur sont retournés. Si le paramètre `limitKind` a la valeur d'un type d'objet de dépôt spécifique, seuls les objets de dépôt de ce type sont renvoyés. Le paramètre `excludeInherited` permet de spécifier si les objets de dépôt qui sont contenus dans le conteneur par héritage sont, ou ne sont pas, renvoyés par la fonction.

La fonction `lookupName` permet de localiser un objet de dépôt, grâce à son nom, dans un conteneur spécifique ou dans les objets de dépôt contenus par ce conteneur. Elle permet, grâce au paramètre `level2Search`, d'indiquer la profondeur de la recherche. Lorsque ce paramètre vaut 1 alors la recherche est cantonnée à l'objet de dépôt du conteneur courant, lorsqu'il est supérieur à 1, la recherche s'effectue récursivement (la profondeur de récurrence est définie par la valeur exacte du paramètre) dans les conteneurs contenus dans le conteneur courant. Une valeur de -1 ne limite pas la profondeur de la recherche. Les paramètres `limitKind` et `excludeInherited` ont la même sémantique que pour la fonction `contents`.

La fonction `describeContents` combine la fonction `contents` et `describe` (définie par l'interface `Contained`). Pour chaque objet de dépôt renvoyé par la fonction `contents`, la description de l'objet est renvoyée (c'est-à-dire que la fonction `describe` est appelée et le résultat renvoyé).

9.4.3 Interface de description du dépôt des schémas d'interactions

`Repository` est une interface qui fournit un accès à l'ensemble des objets de dépôt du dépôt de schémas d'interactions. Un objet de dépôt `Repository` est un conteneur à dossiers. Étant hérité de l'interface `Container`, il peut être utilisé pour rechercher n'importe quel objet de dépôt (et donc n'importe quelle méta-information) défini dans le dépôt de schémas d'interactions par son nom ou son type. L'interface `Repository` est présentée par la figure 9.13.

La fonction `foldersContents` renvoie la liste des objets de type dossiers contenus dans le dépôt de schémas d'interactions. Elle est équivalente à l'appel de la fonction `contents` avec la valeur `isrFolder` pour le paramètre `limitKind` et `TRUE` pour le paramètre `exclureInherited`.

La fonction `foldersDescribeContents` est la restriction de la fonction `Container::describeContents` sur les objets de dépôt de type `isrFolder`.

```
 1:  module CosInteraction
 2:  {
 3:    interface Folder;
 4:
 5:    interface Repository : Container
 6:    {
 7:      // read interface
 8:      ContainedSeq foldersContents ();
 9:      DescriptionSeq foldersDescribeContents (in long maxItems);
10:
11:      // write interface
12:      Folder createFolder (in Identifier name) raises (CANNOT_PROCEED);
13:      InteractionScheme bind (inIdentifier name, in string ISLcode) raises (CANNOT_PROCEED);
14:    };
15:  };
```

FIG. 9.13 – *Interface de description du dépôt de schémas d'interactions*

La fonction `createFolder` renvoie un nouveau dossier vide. Si un dossier du même nom existe déjà l'exception `CANNOT_PROCEED` est déclenchée avec `nameAlreadyExist` comme raison. Le dossier pourra être rempli grâce à certaines fonctions définies dans l'interface `Folder` (décrite ci-après) ou grâce à la fonction `Contained::move`.

9.4.4 Interface de description des dossiers

`Folder` est une interface qui fournit les opérations nécessaires à la gestion des objets de dépôt représentants les dossiers. Un dossier peut contenir d'autres dossiers et des schémas d'interactions. Héritant des interfaces `Repository` et `Contained`, un objet de dépôt représentant un dossier peut être utilisé pour naviguer à travers l'arborescence hiérarchique des dossiers ainsi que pour rechercher les objets de dépôts représentant les schémas d'interactions définis dans le dossier (ou l'un de ses sous-dossiers). L'interface `Folder` est présentée par la figure 9.14.

La fonction `schemesContents` renvoie la liste des objets de type schéma d'interactions contenus dans le dossier. Elle est équivalente à l'appel de la fonction `contents` avec la valeur `isrInteractionScheme` pour le paramètre `limitKind` et `TRUE` pour le paramètre `exclureInherited`. La fonction `schemesDescribeContents` est la restriction de la fonction `Container::describeContents` sur les objets de dépôt de type `isrInteractionScheme`.

La fonction `createScheme` renvoie un nouveau schéma d'interactions vide. Si un schéma d'interactions du même nom existe déjà l'exception `CANNOT_PROCEED` est déclenchée avec `nameAlreadyExist` comme raison. Les méta-informations du schéma d'interactions nouvellement créé pourront être remplies grâce à certaines fonctions définies dans l'interface `InteractionScheme` ou grâce à la fonction `Contained::move`. La

```
 1:  module CosInteraction
 2:  {
 3:    interface Folder : Repository, Contained
 4:    {
 5:      // read interface
 6:      ContainedSeq schemesContents ();
 7:      DescriptionSeq schemesDescribeContents (in long maxItems);
 8:
 9:      // write interface
10:      InteractionScheme createScheme (in Identifier name) raises (CANNOT_PROCEED);
11:      InteractionScheme duplicateScheme (in Identifier name, in InteractionScheme scheme)
                                          raises (CANNOT_PROCEED);
12:    };
13:
14:    struct FolderDescription
15:    {
16:      Identifier name;
17:      Login      owner;
18:    };
19:  };
```

FIG. 9.14 – *Interface de description des dossiers*

fonction duplicateScheme est similaire à la précédente mais les méta-informations du schéma d'interactions créé sont remplies avec celles de l'objet de dépôt du schéma d'interactions fourni en paramètre.

L'appel de la fonction describe, héritée de l'interface Contained, sur un objet de dépôt représentant un dossier retourne un objet de type FolderDescription dans le champ value de la structure Description.

9.4.5 Interfaces de description des opérateurs réactifs

Chaque comportement réactif dispose de sa propre interface pour décrire ses méta-informations dans le dépôt de schémas d'interactions. À l'opposé des interfaces des dossiers ou des schémas d'interactions qui définissent une structure de contenant / contenu, un comportement réactif définit une structure d'arbre (celui de la syntaxe abstraite). Ainsi, une interface de base à tous les comportements réactifs a été définie.

Celle-ci offre à toutes les interfaces des comportements réactifs les deux fonctions suivantes :

- la fonction describe renvoie une structure contenant les méta-informations stockées par l'objet de dépôt. La description exacte de la structure associée à chaque type d'objet de dépôt est fournit ci-dessous lors de la définition réelle de l'interface de l'objet de dépôt. Le champ opKind de la structure Description retournée fourni le type de l'objet le plus approprié vis-à-vis des énumérations définies dans OperationKind et le champ value contient, sous la forme d'un objet any, les méta-informations décrites par le type de l'objet de dépôt.
- la fonction duplicate renvoie une copie de l'objet de dépôt. Le type réellement retourné par cette fonction est celui de l'objet sur lequel a été appliquée la fonction (c'est-à-dire une interface héritant de l'interface Operation).

Les IDL des schémas d'interactions et des comportements réactifs sont directement déduites de la syntaxe abstraite d'ISL et sont décrites en annexe A.

9.5 Propositions partiellement mises en œuvres

L'architecture du modèle à interactions distribuées présentée dans le chapitre précédent propose une projection complète du modèle défini dans les chapitres 5 et 6 en tenant compte de toutes les contraintes imposées par les langages cibles (C++ et Java notamment). Cependant, lors de notre mise en œuvre, nous avons délaissé certaines propositions afin de nous focaliser sur d'autres propositions nous paraissant plus essentielles.

9.5.1 Gestion du typage

Du fait que l'on se projette dans un langage compilé fortement typé, la vérification du typage (afin de s'assurer de la cohérence des envois de messages à l'exécution) est une propriété réalisée par le compilateur lors de la compilation. Les comportements réactifs doivent, eux aussi, être soumis à cette vérification. Or, à ce moment du développement, les interactions, et donc les comportements réactifs, n'existent pas encore. La vérification du typage des messages contenus dans les comportements réactifs doit donc être réalisée dynamiquement et ne peut, par conséquent, être déléguée au langage lui-même. De plus, cette vérification devrait être personnalisable afin de permettre, par exemple, la définition de méthodes de délégation (les méthodes contrôlées n'existent pas à priori dans l'interface des objets).

Ainsi, cette vérification est réalisée lors de l'instanciation d'un schéma d'interactions. Il est à noter que, sous certaines conditions, elle pourrait être réalisée au moment de la définition des schémas d'interactions. Dans ces deux cas, cela implique une introspection de l'application (pouvoir connaître à l'exécution l'ensemble des méthodes définies sur un objet) ou du dépôt d'interfaces (afin de connaître ses opérations publiques).

Cette mise en place d'une gestion du typage implique de distinguer trois vérifications distinctes :

1. La vérification de l'existence des messages déclencheurs dans les classes des objets liés
2. La vérification que les messages contenus dans la réaction existent sur les objets.
3. La vérification que les types des participants d'un schéma d'interactions soient compatibles (au niveau du typage du langage) avec ceux déclarés lors de la définition du schéma d'interactions. Dans le cas du langage C++ ceci signifie que chaque participant est du type déclaré dans le schéma d'interations, soit d'un type dérivé (par héritage).

Ainsi, soit lors de l'instanciation de l'interaction, soit lors de l'activation d'une interaction par la méthode `activate`, il est nécessaire de réaliser cette vérification du typage.

A priori, et à condition de disposer d'un mécanisme d'introspection sur la structure des objets (et plus spécifiquement de la liste des méthodes définies sur un objet), les vérifications concernant la cohérence des interactions ne posent pas de problème particulier et peuvent être réalisées lors de la définition d'un schéma d'interactions (introspection sur les classes) ou lors de l'instanciation d'un schéma d'interactions (introspection sur les objets participants).

Malheureusement, certains des langages cibles (tel que C++ par exemple) ne disposent d'aucun réel mécanisme d'introspection disponible lors de l'exécution de l'application. De ce fait, nous n'avons pas mis en œuvre de mécanisme spécifique permettant la vérification de la cohérence des interactions.

Cependant, le métaobjet d'un objet dispose d'une table d'association (*hashtable*) dont la clef est la signature des méthodes. Celle-ci est utilisée par le mécanisme d'évaluation dynamique des messages contenus dans les comportements réactifs (se reporter au paragraphe 9.2). Ainsi, lors de l'exécution d'un comportement réactif, le mécanisme d'évaluation dynamique des messages réalise, de facto, une vérification de la cohérence des méthodes contenues dans les comportements réactifs et qui doivent être exécutées.

NOTE. — Cette solution peut, lorsqu'une règle d'interaction est incohérente (car faisant appel à une méthode n'existant pas), rendre incohérent le système puisqu'une partie seulement de la règle d'interation à exécuter le sera réellement [2].

Personnalisation de la vérification

Il a également été mentionné que la gestion du typage des interactions devrait être personnalisable pour chacune des trois vérifications proposées.

Pour la vérification de l'existence des messages déclencheurs, une méthode devrait être invoquée, lors de l'instanciation du schéma d'interactions, et renverrait la liste de toutes les règles d'interactions dont les messages déclencheurs ne sont pas définis. Ceci permet de personnaliser le cas 1 de la vérification de la cohérence des interactions et, par exemple, de permettre dynamiquement l'ajout de nouvelles méthodes à un objet [3] (méthodes de délégation).

Pour le test des méthodes invoquées par les comportements réactifs (cas 2), la solution est moins immédiate car les méthodes non définies dans l'interface de l'objet peuvent être des méthodes de délégation définies par d'autres interactions. Ainsi cette vérification s'apparente à la détection dynamique du type des objets (ce qui est aussi le cas pour le cas 3).

9.5.2 Critère de non discrimination des objets

Le modèle à interactions défini au chapitre 5 permet de faire interagir de manière distribuée aussi bien des objets prévus pour être distribués que des objets non prévus pour cela (prise en compte du critère C3.2 de non discrimination des objets pour le support des interactions). Or si un objet non distribué doit être accessible depuis un site distant (par exemple si l'on veut lui envoyer un message) il est nécessaire de rendre cet objet accessible à distance.

Or la mise en œuvre d'un mécanisme générique permettant de manipuler de manière strictement identique les objets distribués et les objets non prévus pour être distribués est loin d'être aisée. Par exemple, l'invocation d'une méthode nécessite de lui transmettre ses paramètres. Dans le cas d'un objet distant, ceci implique que ces derniers transitent par le bus réseau de CORBA. Or seuls les types de base (entiers, chaînes de caractères, flottants, etc.) et les objets distribués savent être traités par CORBA. De plus, pour tout autre type de données, seules les personnes utilisant ces données sont en mesure de définir comment elles doivent être transmises par le bus réseau.

Par conséquent, et afin de nous focaliser sur d'autres points sensibles, nous nous sommes restreints, lors de notre mise en œuvre du modèle à interactions distribuées, à la possibilité de faire interagir entre eux soit des objets non prévus pour être distribués, soit des objets distribués, mais pas un panachage des deux types d'objets (nous décrivons dans l'annexe C une solution répondant partiellement à ce problème).

2. Lorsqu'une incohérence est détectée lors de l'exécution d'un comportement réactif le mécanisme d'évaluation dynamique des messages émet une exception.

3. Il est à noter que cette solution de définition de méthodes de délégation soulève un certain nombre de problèmes liés au fait, que d'une certaine façon, l'on change dynamiquement l'interface de l'objet et donc son type.

9.6 Conclusion

Dans ce chapitre, nous venons de montrer comment nous avons mis en œuvre le modèle à interactions distribuées, présenté dans la partie II et dont l'architecture a été définie dans le chapitre 8, dans l'environnement compilé et fortement typé du langage C++ grâce à l'utilisation de différents schémas de conception.

Parmi ces schémas de conception, on peut citer :

– Schémas de conception Stratégie et Constructeur Virtuel pour mettre en œuvre le mécanisme de réification des paramètres et celui de capture des messages.
– Schémas de conception Commande, Constructeur Virtuel et Poids Mouche pour mettre en œuvre le mécanisme de réification des méthodes et le mécanisme d'évaluation dynamique des messages.

Nous avons proposé, dans cette partie, une mise en œuvre modulaire du protocole à métaobjet défini par notre modèle à interactions distribuées. La conception d'un protocole à métaobjet est une tâche ardue et délicate. Ainsi, la mise en œuvre du protocole présenté dans ce mémoire de thèse est issu de plusieurs définitions successives du protocole, assurant par là même sa maturation.

Par conséquent, nous sommes conscients que ce protocole et, par voie de fait, sa mise en œuvre possède des limites. Nous montrons, dans la partie qui suit, que cette mise en œuvre est cependant suffisamment évoluée pour permettre la réalisation d'applications grandeurs natures : nous présentons un système à base de connaissances.

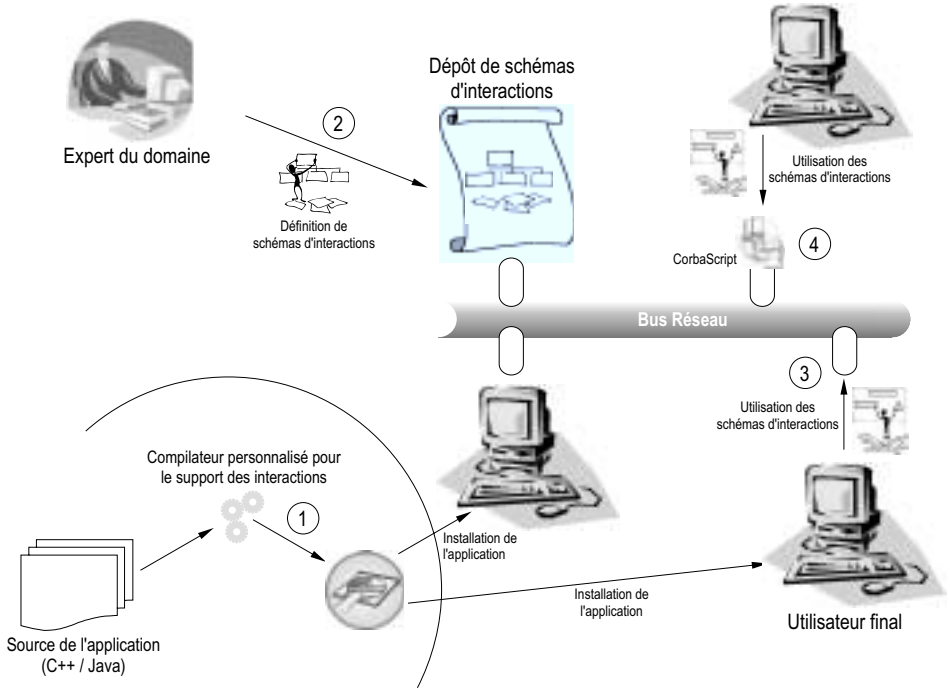


FIG. 9.15 – *Récapitulatif de l'architecture C++ / CORBA*

La figure 9.15 résume comment le mécanisme de capture des messages (point 1), la définition et l'enregistrement des schémas d'interactions (point 2) ainsi que l'utilisation de ces mêmes schémas d'interactions (point 3) se positionnent par rapport à l'architecture globale du système.

CORBAScript est un langage qui peut être utilisé pour accéder au dépôt de schémas d'interactions (point 4), comme cela a été le cas avec les exemples du chapitre 4. En effet, ceci permet, de manière extrêmement simplifié pour l'utilisateur, de définir ou d'instancier des schémas d'interactions ainsi que de parcourir le dépôt de schémas d'interactions (concept d'introspection).

NOTE. — Même si cela n'est pas représenté sur la figure 9.15 un utilisateur final peut également être un expert et définir des schémas d'interactions qu'il enregistre auprès du dépôt de schémas d'interactions.

Partie IV

APPLICATION

Dans cette partie vous trouverez ...

Cette dernière partie présente une application des interactions.

Le chapitre qui la compose (**chapitre 10**) conclut ce mémoire de thèse par une présentation de l'application distribuée développée dans le cadre du projet COLOR [1]. Dans cette application, les interactions ont été utilisées pour interfacer son système à base de connaissances (écrit en C++) à son interface de visualisation (écrite en Java). Il présente en particulier une mise en œuvre du schéma de conception [GHJ+95] *Observateur* à l'aide d'interactions et un début de solution permettant un accès à distance à un objet non destiné à être distribué (c'est-à-dire, dans l'environnement CORBA, un objet n'ayant pas d'interface IDL). Celle-ci est basée sur un mécanisme permettant de nommer les objets non distribués afin qu'ils puissent être manipulés comme les objets distribués.

1. http://www-sop.inria.fr/COLOR/index.html

Chapitre 10

Intégration d'interactions dans un système à base de connaissances

Les systèmes à base de connaissances, pour le pilotage de codes en particulier, sont amenés à évoluer vers des architectures distribuées. Les applications impliquent, en effet, de plus en plus de participants (inter ou intra-entreprises) sur divers sites. L'équipe ORION de l'INRIA Sophia-Antipolis travaille sur cette évolution. Dans le cadre d'un financement COLOR, un partenariat avec le projet ORION a permis de montrer l'intérêt d'utiliser les interactions sur une application en grandeur nature. Ce chapitre est basé sur l'article [PFM01].

10.1 Problématique

10.1.1 Présentation des systèmes à base de connaissances

La notion de générateur de systèmes à base de connaissances a émergé au cours de la seconde moitié des années 80, pour prendre en compte des similarités de besoins. Un système à base de connaissances constitue l'une des nombreuses branches du domaine de l'Intelligence Artificielle (IA). Un générateur permet de réutiliser des éléments communs, comme un moteur d'inférences général, des modes de représentation des connaissances, ainsi que des interfaces graphiques ou des modules d'aide. La similarité de besoins apparaît clairement dans la conception des interfaces graphiques. Les besoins communs sont la *trace* graphique de l'évolution de la connaissance en cours d'exécution et le *contrôle* de certains éléments de connaissances (modification de valeurs par un utilisateur).

L'aspect modulaire d'un système à base de connaissances (figure 10.1) conduit naturellement à vouloir distribuer ses différents composants. Par exemple, ces différents composants peuvent être délocalisés pour des raisons de performances ou de spécificités de l'architecture matérielle et partagés entre les utilisateurs qui doivent souvent collaborer à la mise au point d'un même système à base de connaissances.

Un système à base de connaissances est constitué des éléments suivants.

- Une **base de connaissance**.
  Celle-ci contient une description de la connaissance du domaine : y sont décrits les concepts fondamentaux et leurs propriétés, les relations existantes entre ces concepts ainsi que des règles de résolution d'un ou de plusieurs problèmes types du domaine.
- Une **base de faits**.
  Elle constitue la « mémoire » du système et contient, entre autres, l'état courant du raisonnement. Au départ, elle dispose des données initiales du problème à résoudre.

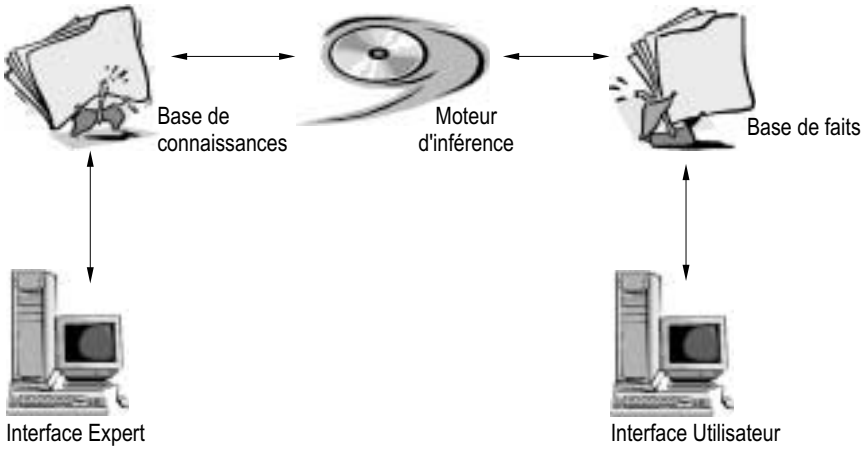


FIG. 10.1 – *Vue d'ensemble d'un système à base de connaissances*

– Un **moteur d'inférences**.
  Il peut être considéré comme le « cerveau » du système et est décomposable en sous-parties. C'est lui qui exploite la connaissance contenue dans la base de connaissances ainsi que les données du problème à résoudre afin de faire évoluer la base de faits vers la solution. Pour ce faire il met en œuvre des mécanismes généraux de combinaison des connaissances.
– Un **ensemble d'interfaces**.
  Elles permettent aux utilisateurs et aux experts de communiquer respectivement avec la base de faits et la base de connaissances.

10.1.2 Tracer et contrôler à distance les systèmes à base de connaissances

Dans le cas d'un expert qui veut modifier le comportement du système à base de connaissances au cours de son exécution, sa connaissance sur le contrôle intervient lors de la communication avec le système. De ce fait, la distribution est elle-même porteuse de connaissances liées au système et à son utilisation. Or actuellement cette connaissance est elle-même distribuée en termes de mise en œuvre technique et souvent difficile à exprimer, à comprendre et à faire évoluer.

L'objectif de l'application consiste à faciliter l'expression et la réutilisation de la connaissance de communication entre un système à base de connaissances et des applications clientes (notamment des interfaces graphiques). Les principaux besoins répertoriés sont l'accès à une base de connaissances à distance pour observer son comportement et/ou le contrôler, la modification dynamique de ce contrôle, le partage d'une base de connaissances entre plusieurs utilisateurs.

Dans un tel type d'application, bien que le contrôle soit en grande partie géré par le système à base de connaissances, il l'est aussi par l'utilisateur. En phase de résolution, les faits et règles du système à base de connaissances dirigent le moteur d'inférence, mais tout expert intéressé par une exécution peut aussi demander à être notifié lors de l'avènement de certains faits. Chaque utilisateur joue donc un rôle de « client » vis-à-vis du système à base de connaissances.

Outre cet aspect de trace, un utilisateur peut également influer sur le comportement du système à base de connaissances en introduisant des « interactions » entre des éléments de connaissances. Ainsi, le contrôle évolue dynamiquement au cours de l'exécution, passant du système aux utilisateurs, sans que ces changements soient prédéfinis. Cette adaptation dynamique du contrôle fait partie intégrante de l'objectif (similaire au critère C4 défini dans la partie I).

Cependant, l'utilisateur, qui n'est en général pas informaticien, doit pouvoir exprimer facilement à quels éléments d'une base de connaissances il veut accéder et quel type de contrôle il veut établir (critère C3).

Les connaissances « relationnelles » que sont la notification et le contrôle ne font pas partie de la connaissance du système. De plus, leur évolution constante interdit l'introduction du contrôle directement dans les entités communicantes, sous peine de devoir régulièrement reconfigurer les systèmes à base de connaissances et les applications clientes (critère C1).

Une même base de connaissances peut être utilisée simultanément par plusieurs utilisateurs. Ainsi, étant donné le caractère très collaboratif du travail du système à base de connaissances, un contrôle centralisé ou dégradant les performances ne peut pas être retenu. L'objectif est donc de distribuer cette connaissance de communication pour que sa prise en compte soit localisée sur les entités à contrôler.

Ceci nécessite une gestion de la collaboration car les contrôles simultanés ne sont pas toujours compatibles. Il s'agit donc, tout en autorisant une distribution du contrôle, de maintenir la cohérence globale de l'application.

10.1.3 Une application de distribution des connaissances

Dans le cadre du développement d'un environnement générique pour la conception de générateurs de systèmes à base de connaissances, nous nous sommes restreint à la réalisation d'un constructeur d'interfaces graphiques adaptable par les utilisateurs de systèmes à base de connaissances.

Les systèmes à base de connaissances construits sont basés sur un *framework* de composants extensibles et adaptables nommé BLOCKS [Moi01]. La base de connaissances des systèmes à base de connaissances générés à partir de ce framework sont écrites en C++ [Str91], dont les attributs sont eux-mêmes réifiés et possèdent une API simple (il s'agit de deux méthodes prédéfinies, l'une pour obtenir le contenu de l'attribut, l'autre pour le modifier) mais générique en fonction du type de l'attribut. Les types des attributs ne sont en effet pas connus à l'avance.

Approche proposée

L'objectif est donc d'enrichir l'environnement générique BLOCKS par des « constructeurs d'interfaces graphiques » qui permettent de construire des interfaces graphiques dédiées et adaptables à l'exécution. Ces constructeurs sont mis à disposition des concepteurs de systèmes à base de connaissances. Ceux-ci peuvent alors choisir d'autoriser les experts ou les utilisateurs finaux à construire dynamiquement leur propre interface pour tracer et contrôler le système à base de connaissances.

Pour ce faire, nous proposons d'établir, en dehors de la base de connaissances, des connexions entre un attribut d'objets d'un système à base de connaissances et des représentations graphiques. Cette représentation graphique est choisie dynamiquement en fonction des besoins de trace et de contrôle, mais aussi en fonction du type de l'attribut. Elle est écrite en Java (traditionnellement utilisé pour les applications sur Internet) et basée sur un noyau graphique (AWT ou Swing). Il s'agit donc de gérer une communication entre des objets C++ (définis avec BLOCKS) et des objets Java.

Comme nous l'avons montré dans la partie I, la connaissance de communication est relativement difficile à mettre en œuvre dynamiquement et à rendre réutilisable de par les limites actuelles des technologies de la distribution. Les interactions présentées dans ce mémoire de thèse répondent aux besoins soulignés précédemment. D'ailleurs ces besoins correspondent aux critères définis dans la partie I et pris en compte par notre modèle à interactions distribuées.

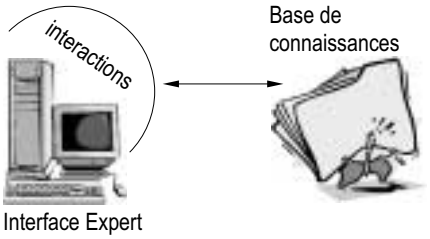


FIG. 10.2 – *Introduction des interactions dans un système à base de connaissances*

Cependant, aucune tentative d'utilisation des interactions pour décrire la communication entre les divers constituants d'un système à base de connaissances n'a été entreprise. L'objectif de cette application est donc double. D'une part, il s'agit de montrer que l'utilisation des interactions dans un système à base de connaissances est possible et, d'autre part, d'intégrer les interactions dans une base de connaissances déjà existante.

Pour ce faire, les interactions vont être introduites entre les interfaces et la base de connaissances afin que les interfaces puissent tracer les objets contenus dans la base de connaissances (figure 10.2). De plus, afin d'adapter dynamiquement les observations sur ces objets, ces interactions pourront être modifiées, supprimées et ajoutées par l'utilisateur via les interfaces.

10.2 Présentation de l'application de construction d'interfaces graphiques

Un utilisateur de systèmes à base de connaissances a souvent besoin de suivre l'évolution d'une base de connaissances lors de son exécution afin de comprendre le cheminement du raisonnement. Ceci est réalisé par la définition de schémas d'interactions. Or l'écriture de ceux-ci à l'aide d'ISL ainsi que leur instanciation sont des opérations complexes pour des non informaticiens que sont généralement les concepteurs de systèmes à base de connaissances et leurs utilisateurs. C'est pourquoi une aide est fournie à ces deux types d'utilisateurs sous la forme de deux bibliothèques.

Nous présentons tout d'abord la structure interne de la base de connaissances. Ensuite nous présentons la bibliothèque de schémas d'interactions générique. Celle-ci est principalement utilisée par les concepteurs de systèmes à base de connaissances pour tracer et contrôler n'importe quel système à base de connaissances. Pour finir, nous décrivons la génération d'une bibliothèque de schémas d'interactions dédiée à un système à base de connaissances spécifique et destinée aux utilisateurs pendant l'exécution.

10.2.1 Analyse de la base de connaissances

Les structures servant à construire la base de connaissances sont issues de BLOCKS afin d'offrir une interface homogène à tous les objets la composant. Ces structures sont au nombre de deux : une classe, nommée `Frame`, héritée par toutes les classes des objets de la base de connaissances, et une classe décrivant les attributs, la classe `Slot`. Voici un exemple d'une classe pour un objet de la base de connaissances :

```
class Car : public Frame
{
  Slot < Car, float > speed;
  Slot < Car, char * > registration;
}
```

Les instances de la classe `Slot` sont comparables à des attributs. D'ailleurs cette classe possède, entre autres, deux méthodes jouant le rôle, pour l'une, de l'accesseur en lecture et, pour l'autre, de l'accesseur en écriture :

```
template <class F, class T> class Slot
{
  setValue (T&);
  T& getValue () const;
  ...
}
```

La visualisation des comportements des objets de la base de connaissances consiste à observer les modifications de l'état de ces objets. Or cet état est intégralement défini par la valeur des attributs de l'objet. De ce fait, ce sont uniquement sur les deux accesseurs des attributs (des instances de la classe `Slot`) que les interactions vont être définies (ceci permet en effet d'exprimer l'ensemble des besoins de l'utilisateur ou de l'expert). Par conséquent, les objets instances de la classe `Slot` sont les seuls objets de la base de connaissances qui vont participer aux interactions.

10.2.2 Bibliothèque générique : un outil d'aide aux concepteurs de systèmes à base de connaissances

Le concepteur d'un système à base de connaissances définit des schémas d'interactions qu'il mettra à la disposition des utilisateurs finaux.

EXEMPLE. — Dans notre cas, il s'agit de choisir les éléments graphiques de représentation des attributs et des schémas d'interactions adaptés à une interface de trace et de contrôle.

Or il existe des schémas d'interactions qui ne diffèrent que par le type de leurs paramètres, en particulier la valeur des attributs. Afin de faciliter la création de tels schémas d'interactions, une bibliothèque générique contenant un ensemble minimal de *schémas d'interactions générique* est proposée.

Chaque schéma d'interactions générique est paramétré par un type de valeur. Il peut générer plusieurs schémas d'interactions différents.

Les schémas d'interactions générique sont décrits à l'aide d'une version étendue du langage ISL afin de supporter le concept de schémas d'interactions paramétrés. Dans l'application développée, les schémas d'interactions génériques concernent les connexions entre attributs d'objets de la base de connaissances et éléments graphiques.

Ainsi, les schémas d'interactions génériques sont une extension des schémas d'interactions. Ils permettent de paramétrer des schémas d'interactions par le type des participants. Ils sont donc l'équivalent, vis-à-vis des schémas d'interactions, des classes génériques (*template classes*) vis-à-vis des classes pour C++.

EXEMPLE. — Voici un exemple de bibliothèque générique contenant cinq schémas d'interactions génériques.

```
GIP1: generic interaction pattern Trace (Attribute<T> attribut, GraphicObject g)
      {
        attribut.setValue (val) -> attribut.setValue (val) ; g.update (val)
      }

GIP2: generic interaction pattern OutOfBounds (Attribute<T> attribut, T bound, GraphicObject g)
      {
        attribut.setValue (val) -> attribut.setValue (val) // if (val > bound) then g.paint ("red") endif
      }

GIP3: generic interaction pattern ConditionalTrace (Attribute<T> attribut, T limit, GraphicObject g, Log l)
        extends Trace (attribut, g)
      {
        attribut.setValue (val) -> if (val < limit) then delegate l.memorize (val) else super endif
      }

GIP4: generic interaction pattern TraceAndControl (Attribute<T> attribut, ModifiableObject g)
      {
        attribut.setValue (val) -> attribut.setValue (val) ; g.update (val),
        g.setValue (val) -> g.setValue (val) // attribut.setValue (val)
      }

GIP5: generic interaction pattern Avoid (Attribute<T> attribut)
      {
        attribut.setValue (val) -> exception reject
      }
```

Le schéma d'interactions générique `GIP1` permet de visualiser un attribut quand sa valeur est modifiée.

Le schéma d'interactions générique `GIP2` exprime une valeur hors limite.

Le schéma d'interactions générique `GIP3` étend le schéma d'interaction générique `GIP1` en définissant un message de délégation.

Le schéma d'interactions générique `GIP4` est un exemple de contrôle sur un attribut via un élément graphique. Une modification de l'entité graphique entraîne l'affectation de la valeur de l'attribut et vice versa.

Enfin, le schéma d'interactions générique `GIP5` permet d'interdire des modifications en levant une exception ce qui empêchera toute autre réaction à la réception de ce message.

La bibliothèque générique est commune à tous les systèmes à base de connaissances et ne dépend pas de l'application. Elle repose uniquement sur les accès aux attributs dans BLOCKS, sur une abstraction d'un noyau graphique d'éléments standards, basés sur Java AWT et sur quelques entités utilitaires (telles que `Log`, `Vector` ou `Listener`) ou des exceptions (telles que `reject`).

À ce niveau abstrait, les schémas d'interactions génériques manipulent de simples « interfaces génériques » (comme `GraphicObject`, `ModifiableObject` ou `Attribute<T>`), dont seuls le nom et l'arité des méthodes applicables (comme `update` pour `GraphicObject`) sont connus.

NOTE. — Le contenu de la bibliothèque générique résulte d'une analyse des besoins communs en trace et contrôle des utilisateurs de systèmes à base de connaissances. Cette bibliothèque générique est le premier élément d'une construction d'interfaces graphiques. Elle est fournie aux concepteurs de systèmes à base de connaissances et peut être étendue pour s'adapter à de nouveaux besoins.

10.2.3 D'une bibliothèque générique à une bibliothèque dédiée à un système à base de connaissances

À partir de la bibliothèque générique, des bibliothèques dédiées de schémas d'interactions pour les utilisateurs finaux vont être produites afin de connecter un système à base de connaissances donné à une interface graphique précise. Une bibliothèque regroupant les schémas nécessaires à la trace et au contrôle d'un système à base de connaissances constitue le second élément d'un constructeur d'interfaces. Elle est fournie aux utilisateurs du système à base de connaissances.

EXEMPLE. — Supposons que l'on veuille tracer la valeur d'un attribut de type réel (`float`) par une jauge. Pour ce faire il faut utiliser l'instruction suivante (proposée par le générateur d'interfaces graphiques) :

```
generate ("Trace", "Attribute<float>", "Gauge", "trace1")
```

Le premier argument est le nom du schéma d'interactions générique à partir duquel le schéma d'interactions est généré. Le deuxième argument est le type de l'attribut tracé, le troisième est le nom de la classe graphique utilisée pour la visualisation et le dernier est le nom du schéma d'interactions à générer. `Gauge` est une projection de l' « interface générique » `GraphicObject` utilisée par le schéma d'interactions générique `Trace`, avec le type réel.

De même, la création d'un schéma d'interactions qui allume un signal lorsqu'un attribut dépasse une limite (130 dans l'exemple) se fait par l'instruction :

```
generate ("OutOfBounds", "Attribute<float>", 130, "Light", "outOfBounds1")
```

Ces deux instructions génèrent les deux schémas d'interactions suivants :

```
interaction pattern trace1 (Attribute<float> attribut, Gauge g)
{
  attribut.setValue (val) -> attribut.setValue (val) ; g.update (val)
}

interaction pattern outOfBounds1 (Attribute<float> attribut, Light g)
{
  attribut.setValue (val) -> attribut.setValue (val) // if (val > 130) then g.paint ("red") endif
}
```

Pour réaliser simplement la génération des schémas d'interactions, le constructeur d'interfaces propose des outils graphiques qui masquent les détails de mise en œuvre du mécanisme de génération.

La définition des schémas d'interactions peut soit être prévue dans la bibliothèque dédiée, soit intervenir dynamiquement au cours de l'exécution, afin d'introduire des contrôles non prévus par le concepteur du système à base de connaissances. En effet, l'utilisateur peut définir ses propres schémas d'interactions pour gérer des interactions propres à son système à base de connaissances éventuellement en utilisant des schémas d'interactions génériques. Cette tâche est facilitée par le constructeur d'interfaces graphiques.

10.2.4 Résumé : une architecture à trois niveaux

Comme nous venons de le voir, la syntaxe d'ISL a été étendue afin de se rapprocher du langage naturel. Avec la génération des interfaces graphiques est offert un support à la définition de schémas d'interactions par le biais de deux bibliothèques permettant la trace et le contrôle des systèmes à base de connaissances.

L'une, destinée aux concepteurs d'un système à base de connaissances, les aide à définir des schémas d'interactions génériques à toutes les bases de connaissances données, et une seconde, destinée aux utilisateurs finaux, les aide dans l'utilisation des schémas d'interactions issus de schémas d'interactions génériques.

Nous avons par conséquent une architecture à trois niveaux : les schémas d'interactions génériques, les schémas d'interactions et les interactions (figure 10.3).

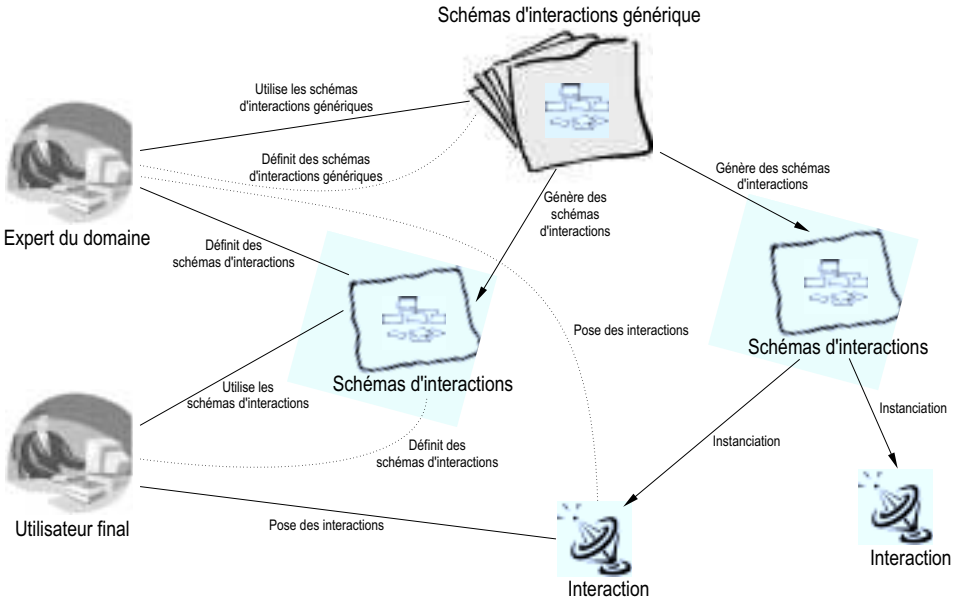


FIG. 10.3 – *Un mécanisme d'interactions à trois niveaux. Les traits pleins décrivent l'usage habituel des experts et des utilisateurs, les pointillés les usages plus anecdotiques.*

10.3 Application : simulation d'un trafic routier

L'architecture décrite ci-dessus a été utilisée afin de développer une interface graphique adaptable permettant de tracer et de contrôler un système à base de connaissances simulant un trafic routier. Dans cet exemple, le système à base de connaissances gère la représentation des véhicules (une classe nommée `Car`) ainsi que de zones de divers types (routes, parking, bas côté, etc.). Le raisonnement du système à base de connaissances détermine leurs déplacements : par exemple, les véhicules ne peuvent pas rouler n'importe où, ou sous certaines conditions (limitation de vitesse notamment). La bibliothèque dédiée a été enrichie avec des schémas d'interactions spécifiques à l'application.

L'interface graphique générée est composée de deux parties : la partie droite est dédiée à la simulation du trafic routier tandis que la partie gauche est dédiée à la modification et à la construction de la partie droite (figure 10.4).

La partie gauche est divisée en trois fenêtres :

- La fenêtre *Frame Chooser* liste les objets du système à base de connaissances qui sont contrôlés. Elle permet à l'utilisateur de sélectionner un attribut à visualiser ou à contrôler par le biais d'un schéma d'interactions. L'utilisateur peut définir une nouvelle interaction à partir d'un schéma d'interactions existant ou afficher l'ensemble des interactions associées à un attribut et, dans ce cas, activer, supprimer ou modifier l'une d'entre elles.
- La fenêtre *Interaction Maker* permet la visualisation des schémas d'interactions et la définition de nouveaux schémas soit à partir d'un schéma d'interactions générique, soit directement à l'aide d'ISL si la bibliothèque générique ne suffit pas.
- La fenêtre *Graphic Object Library* gère les éléments graphiques pertinents pour l'application. L'ajout d'une interaction se fait par un simple clic de souris sur l'objet graphique à connecter à l'attribut sélectionné dans la fenêtre *KB Object Chooser*. L'ajout de nouveaux types de composants graphiques est également possible.

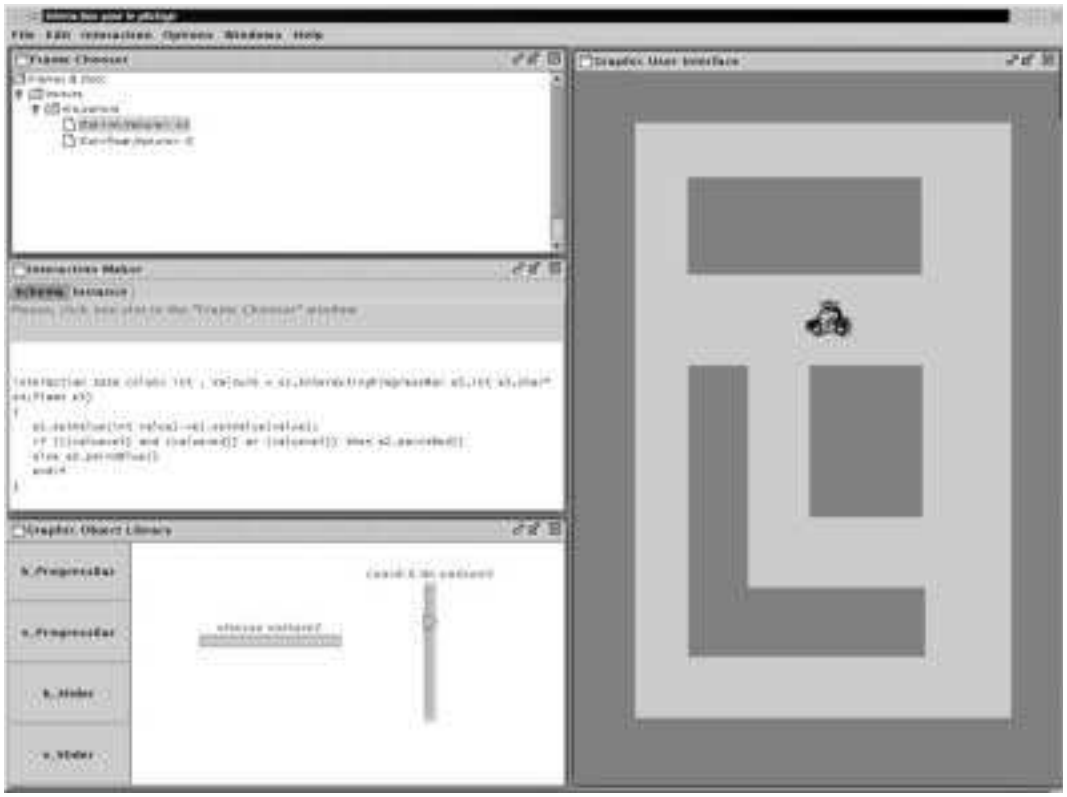


FIG. 10.4 – *Interface graphique de l'application*

La partie droite correspond à l'interface utilisateur. Elle est construite par le concepteur du système à base de connaissances et est utilisée par l'utilisateur final afin de visualiser et contrôler les objets qui l'intéresse.

Dans notre exemple de trafic routier, un véhicule est représenté par une icône qui s'affiche à la position décrite par l'un des objets de la base de connaissances. La couleur de l'icône reflète la vitesse

du véhicule. Ce comportement est spécifié grâce à une interaction. Une réglette permet à l'utilisateur final de changer la vitesse du véhicule.

De même, en déplaçant l'icône, l'utilisateur peut modifier la position du véhicule. Tout ceci est réalisé de manière complètement dynamique durant le raisonnement (grâce au concept des interactions) et ne nécessite aucune modification du code du système à base de connaissances.

10.4 Solution pour la trace des attributs : mise en œuvre du schéma de conception *Observateur*

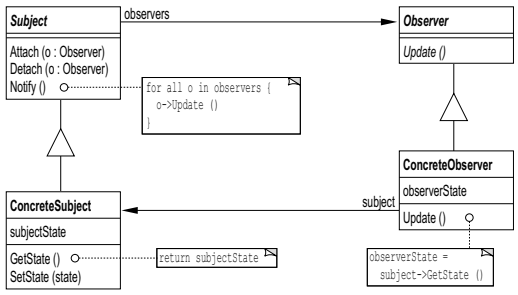
Lors du développement de l'application présentée dans ce chapitre, nous avons mis en œuvre le mécanisme de trace des objets de la base de connaissances grâce au schéma de conception (*design pattern*, [GHJ+95]) *Observateur*. Dans ce paragraphe, nous en présentons une version générale.

Nous montrons, avec ce schéma de conception, comment les interactions sont une solution simple et tout à fait adaptée pour la mise en œuvre de certains schémas de conception.

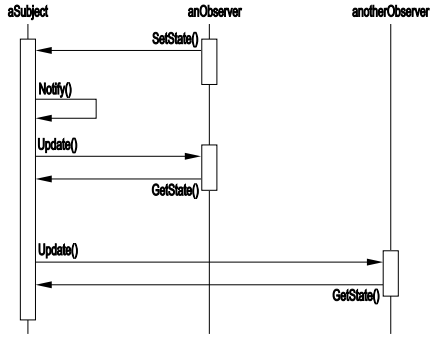
10.4.1 Mise en œuvre du schéma de conception *Observateur* à l'aide d'interactions

RÔLE. — Un schéma de conception comportemental décrit non seulement la composition entre un ensemble de classes ou d'objets mais aussi les schémas de communication entre ces classes ou objets. C'est le cas du schéma de conception *Observateur* dont le but est de définir une relation entre un ensemble d'objets de telle façon que la modification de l'état de l'un des objets soit notifiée aux autres objets afin qu'ils puissent s'adapter à ce nouvel état.

SOLUTION CLASSIQUE. — Lorsqu'un objet, appelé le sujet, change d'état, il prévient les objets qui dépendent de lui (les observateurs) grâce à l'invocation de sa méthode [1] `Notify`. Ceci a pour conséquence l'invocation de la méthode `Update` sur chacun des observateurs (figure 10.5b). La structure classique de ce schéma de conception est présentée par la figure 10.5a (issue de [GHJ+95]).



10.5a – Schéma de conception Observateur



10.5b – Scénario illustrant la collaboration entre un sujet et deux observateurs

FIG. 10.5 – *Schéma de conception* Observateur

NOTRE APPROCHE. — Lorsqu'un objet est observé, la sémantique de ces méthodes modifiant son état est modifiée afin de tenir compte des observateurs. Or c'est le but premier des interactions. Ainsi, nous modélisons un tel schéma de conception par un schéma d'interactions dont le rôle est d'invoquer automatiquement une ou plusieurs méthodes sur les observateurs afin de les mettre à jour. De ce fait, la présence des méthodes `Notify` et `Update` n'est plus requise puisque la sémantique de mise à jour est intégralement contenue et exprimée dans l'interaction par le biais des méthodes modifiant l'état de l'objet observé.

Afin d'illustrer notre approche et de pouvoir la comparer à la solution traditionnelle, nous reprenons l'un des exemples d'illustration du schéma de conception Observer [GHJ+95, pp. 293–303], à savoir l'exemple de l'horloge digitale. Nous donnons ci-après le code complet de cet exemple tel qu'il est présenté dans [GHJ+95] (le code sur fond grisé correspond au code spécifique à la mise en œuvre du schéma de conception).

1. Le nom de cette méthode, et de celle pour la mise à jour des observateurs, est celui proposé par les auteurs de [GHJ+95].

```
class Subject;

class Observer {
  public:
    virtual ~Observer();
    virtual void Update(Subject* theChangedObject) = 0;
  protected:
    Observer();
};

class Subject {
  public:
    virtual ~Subject();

    virtual void Attach(Observer* o)
      { _observers->Append(o); }
    virtual void Detach(Observer*)
      { _observers->Remove(o); }

    virtual void Notify() {
      ListIterator<Observer*> i(_observers);

      for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Update(this);
      }
    }
};
```

```
class ClockTimer : public Subject {
  public:
    ClockTimer ();

    virtual int GetHour();
    virtual int GetMinute();
    virtual int GetSecond();

    void Tick() {
      // update internal time-keeping state
      // ...
      Notify();
  }
};

class DigitalClock : public Widget, public Observer {
  public:
    DigitalClock (ClockTimer* s) : _subject(s)
      { _subject->Attach(this); }
    ~DigitalClock()
      { _subject->Detach(this); }

    virtual void Update(Subject* theChangedObject) {
      if (theChangedSubject == _subject)
        Draw();
    }

    virtual void Draw() {
      int hour = _subject->GetHour();
      int minute = _subject->GetMinute();
      // draw the digital clock
    }

  private:
    ClockTimer* _subject;
};
```

Nous proposons maintenant une première version de cet exemple à l'aide des interactions. Cette version est une traduction immédiate du schéma de conception défini dans [GHJ+95] :

```
interaction Observer_v1 (ClockTimer timer, DigitalClock clock)
{
  timer.Tick () -> timer.Tick () ; clock.Update (timer)
}
```

Nous pouvons noter, sur l'exemple issu de [GHJ+95] ou sur la version à interactions ci-dessus, que les objets observés (les sujets) et les objets observateurs doivent être préparés en vue d'être observés ou observateurs (par héritage soit de la classe `Subject`, soit de la classe `Observer`). De ce fait, dès la phase de conception de l'application le programmeur doit déterminer quelles sont les classes qui pourront être observées.

Afin d'éviter ce problème, nous proposons une seconde version du même exemple ne nécessitant pas aux classes `ClockTimer` et `DigitalClock` d'hériter respectivement des classes `Subject` et `Observer` et donc d'en supporter l'interface (ces deux classes n'incluent donc pas le code en grisé de l'exemple ci-dessus) :

```
interaction Observer_v2 (ClockTimer timer, DigitalClock clock)
{
  timer.Tick () -> timer.Tick () ; [ clock.SetHour (timer.GetHour ()) //
                                     clock.SetMinute (timer.GetMinute ()) ] ; clock.Draw ()
}
```

10.5 Conclusion

Dans ce chapitre nous avons montré que les interactions sont un outil de mise en œuvre adapté pour la construction de systèmes à base de connaissances distribués. En effet, les interactions répondent aux différents besoins d'une telle application. Elles assurent la séparation logique entre la base de connaissances et les interfaces graphiques, la prise en compte du caractère éminemment dynamique et coopératif de la communication entre objets de la base de connaissances et l'interface graphique, ainsi que la cohérence de cette communication.

CONCLUSION GÉNÉRALE

Chapitre 11

Conclusion générale

Dans ce mémoire de thèse, nous nous sommes intéressés à la conception d'un modèle à interactions distribuées et à sa mise en œuvre dans un environnement compilé et fortement typé. Nous avons précisé les propriétés devant être prises en compte pour un support idéal des interactions sous la forme d'un ensemble de critères qui sont vérifiés par notre modèle à interactions distribuées. Nous avons proposé une mise en œuvre de ce modèle par la définition d'un protocole à métaobjets permettant d'offrir aux interactions une solution non seulement extensible, modulaire et adaptable, mais aussi efficace.

11.1 Bilan

Les interactions, et plus particulièrement les comportements réactifs définis par les interactions, sont des fonctionnalités transversales aux concepts de classes et d'objets. Leur définition s'unifie parfaitement à celle d'un aspect [KLM+97]. De plus, l'étude des différentes approches (présentées dans la partie I) offrant un support aux interactions a permis de mettre en évidence que le concept d'interactions est indépendant de tout langage de programmation donné. Ainsi, dans le modèle à objets, le concept d'interactions est intimement lié au concept d'envoi de messages (mais non à sa mise en œuvre effective). Par conséquent, il nous a semblé inopportun de définir les interactions dans le langage applicatif. Nous avons donc défini un **langage spécifique pour décrire les interactions**, que nous avons nommé *Interaction Specification Language* (ISL).

Ainsi, les programmeurs définissent, dans le langage applicatif, les classes et *définissent*, dans le langage ISL, des schémas d'interactions. Ensuite ils peuvent créer des *instances* de ces schémas d'interactions, les interactions, en déclarant les objets participants à ces interactions. Nous avons montré que, grâce à ISL, la définition et l'instanciation des schémas d'interactions sont réalisées sans modifier les classes des objets participants aux interactions. En effet, **les schémas d'interactions et les interactions sont dissociés des classes et des objets** : la *définition* d'un schéma d'interactions n'est liée d'aucune manière à celle d'une classe et l'*instanciation* d'un schéma d'interactions n'est liée à aucune instanciation d'objet. Ceci permet d'instancier un schéma d'interactions bien après l'instanciation des objets qui vont y participer et de supprimer cette interaction bien avant la destruction des objets participants.

Le caractère des interactions est éminemment dynamique. Ainsi, nous avons montré l'importance de l'adaptation dynamique de notre modèle à interactions distribuées et de sa mise en œuvre. Les contraintes imposées par nos langages cibles, tels que C++ ou Java, nous ont amené à proposer toute une infrastructure disponible à l'exécution pour permettre un support des aspects dynamiques que nous définissons dans notre modèle. Par conséquent, nous avons défini **un protocole à métaobjets disponible à l'exécution** et montré l'importance des apports des concepts issus de la programmation réflexive dans le cadre des architectures adaptables et extensibles dynamiquement. En effet, à notre connaissance, la réflexion est l'unique technique permettant d'assurer une réelle flexibilité à l'exécution (telle que le contrôle sélectif de l'envoi de messages) et une réutilisation maximale des composants.

De plus, la découverte dynamique de nouveaux services est réalisée par les architectures distribuées visées par notre travail grâce au mécanisme de dépôt des services distribués (*service repository*). Ce dépôt de services joue un rôle actif dans la prise en compte d'aspects dynamiques par ces architectures.

Nous avons donc exploité cette idée et défini le concept de **dépôt de schémas d'interactions**. Ce dernier permet de déposer dans une base de données l'ensemble des informations concernant les schémas d'interactions et ne pouvant pas être représentées dans le dépôt d'interfaces de l'architecture distribuée.

Lorsqu'une interaction est déclenchée, cela implique l'exécution du comportement réactif associé au message ayant déclenché l'interaction. Cette exécution nécessite notamment l'envoi de messages aux objets participants à l'interaction. Cet envoi de messages implique l'invocation « physique » des méthodes désignées par ces messages. Puisque nous nous plaçons dans un contexte compilé, il n'est pas possible de simplement évaluer une méthode à partir de son nom, comme cela est le cas dans un contexte interprété. Par conséquent, nous avons défini et mis en œuvre un mécanisme d'**évaluation dynamique des messages** pour les environnements compilés et fortement typés. Ce mécanisme est décrit à l'aide d'un ensemble de schémas de conceptions (*design patterns*) [GHJ+95].

De plus, lorsque plusieurs comportements réactifs sont associés à un même message leur exécution est interdépendante afin de conserver la sémantique des différents comportements réactifs. Nous avons ainsi défini de manière formelle un mécanisme, **la fusion comportementale**, permettant la combinaison de comportements réactifs en vue de leur exécution.

Pour résumer, nous avons montré :

- L'importance d'offrir une réelle abstraction aux interactions. Celle-ci est modélisée par la notion de schémas d'interactions.
- Les apports de la définition d'un langage spécifique pour décrire les schémas d'interactions.
- L'importance de la définition de la fonction de fusion comportementale.
- La nécessité de proposer un système ouvert et adaptable dynamiquement.
- Les apports de la réflexion pour la mise en œuvre de notre modèle.

11.2 Réflexions et travaux futurs

À partir des travaux présentés dans ce mémoire de thèse, nous présentons ci-dessous différents axes de recherche à approfondir :

- **Amélioration du support du distribué**
  Nous avons présenté, dans ce mémoire de thèse, une mise en œuvre possible de notre modèle à interactions distribuées en C++ et CORBA. Nous sommes intimement persuadés que notre concept d'interactions peut être intégré à CORBA sous la forme d'un service. Ceci implique une étude afin de positionner nos interactions vis-à-vis des services déjà existant (et notamment les *Event and Relationship Services*) puis une soumission auprès de l'OMG.

  Des travaux restent à faire autour des propriétés des environnements distribués. En effet, nous pensons que le principe des interactions apporte une solution adaptée pouvant simplifier la description de la sémantique de certaines propriétés chères au distribué. Parmi ces propriétés on peut citer la gestion de la tolérance aux fautes ainsi que le support des communications transactionnelles. L'objectif de ces travaux serait d'étudier les apports d'une définition de bibliothèques dédiées au distribué.

  De plus, il nous semble intéressant d'évaluer comment les interactions peuvent être utilisées dans des environnements à base de composants distribués, tels que les EJB [MH99] ou CCM [OMG99, MM00, MMG+01], pour définir des connecteurs entre les composants logiciels afin d'offrir des architectures plus flexibles et d'un haut degré d'adaptation. Cela pourrait permettre une intégration plus dynamique et ouverte de nouveaux services.

- **Évolution du langage ISL**
  Par définition, le langage ISL se veut indépendant de tout langage de programmation. Cependant, lors du développement multi-langages, il peut arriver qu'un composant ne se nomme pas pareil dans les différents langages. De plus, nous avons introduit dans ISL du typage. Mais la sémantique associée à ce typage n'est pas toujours clairement définie. En particulier quelle est la signification du typage dans un langage tel que Smalltalk, quelle doit être la notion de « typage commun » à l'ensemble des langages dans lesquels ISL peut être projeté ?

  Nous avons également fait en sorte que le langage ISL soit minimal au niveau du nombre d'opérateurs réactifs proposés tout en assurant un fort pouvoir d'expression des comportements réactifs. Cependant l'expression des structures de contrôle avec le langage ISL est quelque peu négligée. Par exemple, le test de comparaison d'un paramètre par rapport à une constante ne peut être écrite `param < 130` mais implique l'invocation d'une méthode contenant ce test.

Lors du développement de l'application présentée dans le chapitre 10, nous nous sommes rendu compte de l'importance de proposer au-dessus de la « brique ISL de base » présentée dans ce manuscrit des bibliothèques dédiées à différents domaines métiers.

Ainsi, des travaux doivent être entrepris afin d'apporter une réelle sémantique du typage à ISL et, pour certains domaines d'applications, d'offrir un pouvoir d'expression plus important et, surtout, moins contraignant vis-à-vis du domaine.

– **D'une fonction de fusion comportementale localisée vers un contrôle global**

Des solutions doivent être trouvées afin d'assurer une meilleure prise en compte de la cohérence globale du graphe des interactions. Ceci implique un ensemble de travaux tournant autour de la fonction de fusion comportementale. En effet, la fonction de fusion comportementale peut conduire actuellement à des échecs uniquement à l'exécution. Il nous semble judicieux de pouvoir détecter sur le graphe des interactions les échecs à l'avance.

De même, une détection des cycles au sein du graphe des interactions doit être proposée, sans pour autant pénaliser l'efficacité de l'architecture. Ces travaux doivent étudier diverses possibilités quant à la détection : une détection statique lors de la définition des schémas d'interactions (afin de déterminer quels sont les schémas d'interactions potentiellement dangereux d'utiliser ensemble), ou une détection dynamique, lors de l'instanciation des schémas d'interactions. Dans le cadre de la détection dynamique, il serait envisageable d'exploiter la possibilité de définir des interactions sur d'autres interactions.

Afin d'améliorer les performances globales de l'application, et en contrepartie de la dynamicité des interactions, nous proposons d'appliquer, lorsque cela est possible, la fonction de fusion comportementale à la compilation et à transcrire en « code pur » le résultat de cette fusion. Cette dernière optimisation ne concerne que les interactions définies statiquement dans le graphe des interactions et dont la durée de vie est identique à celle des objets interagissants. Elle évite, pour ces interactions, la mise en place du mécanisme de capture et d'évaluation dynamique des messages et, par voie de conséquence, la réification des paramètres et des méthodes. Dans le cadre des travaux sur l'étude du graphe des interactions une voie de recherche serait d'étudier un rapprochement vers les mondes du synchrone (et notamment Esterel).

– **Empreinte minimale du modèle à interactions distribuées sur les applications**

Il est primordial pour nous que le mécanisme d'exécution des interactions défini par notre architecture soit efficace en offrant une empreinte sur l'application aussi faible que possible. Cependant nous n'avons pas présenté toutes les techniques connues pour minimiser au maximum cette empreinte. Nous pensons en particulier que la réification des seules méthodes qui feront réellement partie d'au moins une interaction (au lieu de toutes les méthodes publiques) est une optimisation importante car elle peut considérablement réduire le nombre d'objets instanciés par notre architecture pour son propre fonctionnement. Cependant, ceci implique d'avoir connaissance lors de la phase de compilation de l'interface interagissante des classes des objets interagissants [1].

Une autre optimisation importante consiste à disposer d'un *pool* d'objets encapsulant les paramètres des messages capturés. Ceci évite ainsi la création et la destruction (qui sont des opérations très coûteuses) de ces capsules à chaque invocation d'un message. Cette optimisation peut être mise en œuvre à l'aide du schéma de conception Poids Mouche (Flyweight Design Pattern [GHJ+95, pp. 195–206]).

– **Référencement de l'ensemble des objets du système**

Alors que les modèles d'architectures distribuées, tels que CORBA [OMG98] ou Java RMI [Dow98], définissent une notion de référence aux objets distribués, le modèle à objets « conventionnel », qui définit les objets non prévus pour être distribués, n'offre aucun moyen de référencer ces objets.

Notre modèle à interactions distribuées a introduit le concept de référencement à tous les objets du système (qu'ils soient ou non prévus pour être distribués) afin que des interactions distribuées puissent être instanciées entre des objets non prévus pour être distribués. Cependant, notre mise en œuvre du modèle pêche sur ce point puisque nous n'avons proposé qu'une solution répondant partiellement à ce critère (cette solution est décrite en annexe C). Ainsi des travaux restent à faire afin d'apporter une solution répondant entièrement au problème qui soit indépendante d'un support de communication donné.

1. Ceci peut être réalisé par une extension de la syntaxe du langage applicatif (grâce à l'utilisation d'un compilateur ouvert) ou par la définition d'un langage de description d'interfaces similaire au langage IDL défini par CORBA voire, tout simplement, par son utilisation.

11.3 Publications et réalisations

Des publications (rapport des recherches ainsi qu'articles à des groupes de travail et des conférences nationales et internationales) ont porté sur les travaux présentés dans ce mémoire de thèse :

- La mise en œuvre d'un modèle à interactions entre objets distants à base de métaobjets dans des langages objets compilés et fortement typés (C++ [Str91], Java [CH96]),
- L'utilisation de la réflexivité (par le biais d'un MOP statique) pour définir un protocole à métaobjets (MOP) [BKdR91] dynamique spécialisé dans la gestion des interactions [Ber98],
- L'adéquation et les limites de l'approche par Aspects [KLM+97] pour modéliser les interactions [BDF98],
- La réification de l'appel de méthode pour une évaluation dynamique des messages dans un environnement compilé, typé et distribué [Ber99] à l'aide de schémas de conception [GHJ+95],
- L'abstraction de la mise en œuvre des langages d'aspects grâce à la description de la localisation des points d'ancrage (*join points*) à l'aide d'un aspect [Ber00].
- Un état de l'art concernant la prise en compte des interactions par les modèles à objets et à composants [Ber01].

Une version, nommée DICO++, de l'architecture dynamique définie pour le modèle à interactions entre objets distants a été mise en œuvre en C++ [Str91] et CORBA [OMG98]. Une autre version de ce même modèle à interactions a été réalisée en Java (à l'aide de RMI dans un premier temps, puis à l'aide de CORBA dans un second temps), ceci afin de démontrer le support de l'hétérogénéité des langages et des protocoles distribués sous-jacents. Une application (présentée dans le chapitre 10), écrite pour moitié en C++/CORBA et, pour autre moitié, en Java/CORBA a également été développée.

L'ensemble de ces réalisations, les publications portant sur le travail présenté dans ce mémoire de thèse ainsi que des présentations sont disponibles à partir du site Web du projet MICADO à l'adresse `http://micado.project.free.fr`.

Annexes

Annexe A

Interfaces IDL du dépôt de schémas d'interactions

Nous présentons, dans cette annexe, l'interface IDL permettant, pour chacune des constructions du langage ISL, d'en obtenir ou d'en définir les propriétés au sein du dépôt de schémas d'interactions.

A.1 Interfaces des comportements réactifs

Chaque comportement réactif dispose de sa propre interface pour décrire ses méta-informations dans le dépôt de schémas d'interactions. À l'opposé des interfaces des dossiers ou des schémas d'interactions qui définissent une structure de contenant / contenu, un comportement réactif définit une structure d'arbre (celui de la syntaxe abstraite). Ainsi, une interface de base à tous les comportements réactifs a été définie.

A.1.1 Interface commune à tous les comportements réactifs

L'interface de base `Operation` est l'interface commune aux objets de dépôt décrivant les méta-informations des comportements réactifs. Elle est non instanciable. Ainsi tous les objets de dépôt décrivant un comportement réactif héritent de cette interface, présentée par la figure A.1 page suivante.

L'attribut `opKind` identifie le type exact du comportement réactif.

La fonction `describe` renvoie une structure contenant les méta-informations stockées par l'objet de dépôt. La description exacte de la structure associée à chaque type d'objet de dépôt est fournie ci-après lors de la définition réelle de l'interface de l'objet de dépôt. Le champ `opKind` de la structure `Description` retournée fourni le type de l'objet le plus approprié vis-à-vis des énumérations définies dans `OperationKind` et le champ `value` contient, sous la forme d'un objet `any`, les méta-informations décrites par le type de l'objet de dépôt.

La fonction `duplicate` renvoie une copie de l'objet de dépôt. Le type réellement retourné par cette fonction est celui de l'objet sur lequel a été appliquée la fonction (c'est-à-dire une interface héritant de l'interface `Operation`).

A.1.2 Interface de description du comportement réactif d'assignation

`AssignOperation` est une interface qui permet de manipuler les méta-informations associées à la description d'un comportement réactif d'assignation. L'interface `AssignOperation` est présentée par la figure A.2 page suivante.

L'attribut `msg` décrit le message (un comportement réactif d'envoi de messages ou d'assignation) dont le résultat (la valeur de retour du message) doit être assigné à la variable décrite par l'attribut `var`.

```
 1:  module CosInteraction
 2:  {
 3:    enum OperationKind
 4:    {
 5:      opException,
 6:      opCall,
 7:      opAssign,
 8:      opWait,
 9:      opSequence,
10:      opConcurrency,
11:      opIfThenElse,
12:      opTryCatch
13:    };
14:
15:    interface Operation : ISRObject
16:    {
17:      // read interface
18:      readonly attribute OperationKind opKind;
19:
20:      struct Description
21:      {
22:        OperationKind opKind;
23:        any           value;
24:      };
25:
26:      Description describe ();
27:      Operation duplicate ();
28:    };
29:  };
```

FIG. A.1 – *Interface commune à tous les comportements réactifs*

L'appel de la fonction `describe` héritée de l'interface `Operation` sur un objet de dépôt représentant un comportement réactif d'assignation retourne un objet de type `AssignOperationDescription` dans le champ `value` de la structure `Description`. Le champ `msgAssignCall` est rempli par la description du message `msg`. Le contenu exact de ce champ est fonction du type exact de l'attribut `msg` (retourné dans le champ `msgKind`).

```
 1:  module CosInteraction
 2:  {
 3:    interface MsgAssignWaitOperation : Operation {};
 4:    interface MsgAssignOperation : MsgAssignWaitOperation {};
 5:
 6:    interface AssignOperation : MsgAssignOperation
 7:    {
 8:      // read/write interface
 9:      attribute MsgAssignOperation msg;
10:      attribute Identifier         var;
11:    };
12:
13:    struct AssignOperationDescription
14:    {
15:      Identifier   variable;
16:      OperationKind msgKind;
17:      any           msgAssignCall;
18:    };
19:  };
```

FIG. A.2 – *Interface de description du comportement réactif d'assignation*

A.1.3 Interface de description du comportement réactif d'attente

`WaitOperation` est une interface qui permet de manipuler les méta-informations associées à la description d'un comportement réactif d'attente. L'interface `WaitOperation` est présentée par la figure A.3.

L'attribut msg décrit le message (un comportement réactif d'envoi de messages, d'assignation ou d'attente) dont le lancement de l'exécution est contraint par la fin de l'exécution du message décrit par l'attribut waitMsg.

```
 1:  module CosInteraction
 2:  {
 3:    interface WaitOperation : MsgAssignWaitOperation
 4:    {
 5:      // read/write interface
 6:      attribute MsgAssignWaitOperation msg;
 7:      attribute CallOperation          waitMsg;
 8:    };
 9:
10:    struct WaitOperationDescription
11:    {
12:      CallOperationDescription   message;
13:      OperationKind              msgKind;
14:      any                        msgAssignWaitCall;
15:  };
```

FIG. A.3 – *Interface de description du comportement réactif d'attente*

L'appel de la fonction describe, héritée de l'interface Operation, sur un objet de dépôt représentant un comportement réactif d'attente retourne un objet de type WaitOperationDescription dans le champ value de la structure Description. Le champ msgAssignWaitCall est rempli par la description du message msg. Le contenu exact de ce champ est fonction du type exact de l'attribut msg (retourné dans le champ msgKind).

A.1.4 Interface de description du comportement réactif d'envoi de messages

```
 1:  module CosInteraction
 2:  {
 3:    typedef string TypeIdentifier;
 4:
 5:    enum ParameterMode { PARAM_IN, PARAM_OUT, PARAM_INOUT };
 6:
 7:    struct ParameterDescription
 8:    {
 9:      Identifier     name;
10:      TypeIdentifier type;
11:      ParameterMode  mode;
12:    };
13:    typedef sequence <ParameterDescription> ParDescriptionSeq;
14:
15:    struct Message
16:    {
17:      Identifier object;
18:      Identifier method;
19:    };
20:
21:    interface CallOperation : MsgAssignOperation
22:    {
23:      // read/write interface
24:      attribute Message           msg;
25:      attribute ParDescriptionSeq params;
26:    };
27:
28:    struct CallOperationDescription
29:    {
30:      Message           msg;
31:      ParDescriptionSeq parameters;
32:    };
33:  };
```

FIG. A.4 – *Interface de description du comportement réactif d'envoi de messages*

CallOperation est une interface qui permet de manipuler les méta-informations associées à la description d'un comportement réactif d'envoi de messages. L'interface CallOperation est présentée par la figure A.4 page précédente.

L'attribut msg décrit le message qui doit être exécuté. L'attribut params décrit les paramètres du message. Il se compose d'une liste d'éléments de type ParameterDescription. L'ordre des éléments dans la liste est importante et correspond à l'ordre des paramètres du message à exécuter. Le membre name fournit le nom du paramètre tandis que le membre mode indique si le paramètre est un paramètre d'entrée, de sortie ou d'entrée/sortie.

L'appel de la fonction describe, héritée de l'interface Operation, sur un objet de dépôt représentant un comportement réactif d'envoi de messages retourne un objet de type CallOperationDescription dans le champ value de la structure Description.

A.1.5 Interfaces de description des comportements réactifs séquentiel et concurrentiel

BinaryOperation est une interface de base, non instanciable, qui permet de manipuler les méta-informations associées à la description des comportements réactifs comportant deux sous comportements réactifs (cas des comportements réactifs séquentiel et concurrentiel). L'interface BinaryOperation est présentée par la figure A.5.

De cette interface sont dérivées les interfaces permettant de manipuler les méta-informations associées à la description des comportements réactifs séquentiel (interface SequenceOperation) et concurrentiel (interface ConcurrencyOperation).

L'appel de la fonction describe, héritée de l'interface Operation, sur un objet de dépôt représentant soit un comportement réactif séquentiel, soit un comportement réactif concurrentiel retourne un objet de type BinaryOperationDescription dans le champ value de la structure Description. Le champ leftDescription (respectivement rightDescription) renvoie la description du comportement réactif leftPart (respectivement rightPart).

```
 1:  module CosInteraction
 2:  {
 3:    interface BinaryOperation : Operation
 4:    {
 5:      // read/write interface
 6:      attribute Operation leftPart;
 7:      attribute Operation rightPart;
 8:    };
 9:
10:    interface SequenceOperation : BinaryOperation {};
11:    interface ConcurrencyOperation : BinaryOperation {};
12:
13:    struct BinaryOperationDescription
14:    {
15:      any leftDescription;
16:      any rightDescription;
17:    };
18:  };
```

FIG. A.5 – *Interfaces de description des comportements réactifs séquentiel et concurrentiel*

A.1.6 Interface de description du comportement réactif conditionnel

ConditionalOperation est une interface qui permet de manipuler les méta-informations associées à la description d'un comportement réactif conditionnel. L'interface ConditionalOperation est présentée par la figure A.6 page ci-contre.

L'attribut msg décrit le message (un comportement réactif d'envoi de messages) dont le résultat booléen servira de condition pour l'exécution de l'un des deux comportements réactifs décris par les attributs thenPart et elsePart.

```
 1:  module CosInteraction
 2:  {
 3:    interface ConditionalOperation : Operation
 4:    {
 5:      // read/write interface
 6:      attribute CallOperation msg;
 7:      attribute Operation     thenPart;
 8:      attribute Operation     elsePart;
 9:    };
10:
11:    struct ConditionalOperationDescription
12:    {
13:      CallOperationDescription message;
14:      any                      thenDescription;
15:      any                      elseDescription;
16:    };
17:  };
```

FIG. A.6 – *Interface de description d'un comportement réactif conditionnel*

L'appel de la fonction `describe`, héritée de l'interface `Operation`, sur un objet de dépôt représentant un comportement réactif conditionnel retourne un objet de type `ConditionalOperationDescription` dans le champ `value` de la structure `Description`. Les champs `thenDescription` et `elseDescription` renvoient respectivement la description du comportement réactif `thenPart` et `elsePart`. Le champ `message` renvoie la description du comportement réactif de l'attribut `msg`.

A.1.7 Interface de description du comportement réactif d'exception

`ExceptionOperation` est une interface qui permet de manipuler les méta-informations associées à la description d'un comportement réactif d'exception. Elle est présentée par la figure A.7.

```
 1:  module CosInteraction
 2:  {
 3:    interface ExceptionOperation : Operation
 4:    {
 5:      attribute Identifier except;
 6:    };
 7:
 8:    struct ExceptionOperationDescription
 9:    {
10:      Identifier exception;
11:    };
12:  };
```

FIG. A.7 – *Interface de description d'un comportement réactif d'exception*

L'identificateur de l'exception déclenchée par le comportement réactif est décrit par l'attribut `except`.

L'appel de la fonction `describe`, héritée de l'interface `Operation`, sur un objet de dépôt représentant un comportement réactif d'exception retourne un objet de type `ExceptionOperationDescription` dans le champ `value` de la structure `Description`. L'unique champ décrit l'identificateur de l'exception déclenchée.

A.1.8 Interface de description du comportement réactif de traitement des exceptions

`TryCatchOperation` est une interface qui permet de manipuler les méta-informations associées à la description d'un comportement réactif de traitement des exceptions. L'interface `TryCatchOperation` est présentée par la figure A.8 page suivante.

Le comportement réactif décrit par l'attribut `tryPart` est « protégé » contre l'exception dont l'identificateur est spécifiée par l'attribut `except` par le comportement réactif décrit par l'attribut `catchPart`.

L'appel de la fonction `describe`, héritée de l'interface `Operation`, sur un objet de dépôt représentant un comportement réactif de traitement des exceptions retourne un objet de type `TryCatchOperationDescription` dans le champ `value` de la structure `Description`. Les champs `tryDescription` et `catchDescription` renvoient

respectivement la description du comportement réactif tryPart et catchPart. Le champ msg renvoie l'identificateur de l'exception traitée.

```
 1:  module CosInteraction
 2:  {
 3:    interface TryCathOperation : Operation
 4:    {
 5:      attribute Identifier except;
 6:      attribute Operation  tryPart;
 7:      attribute Operation  catchPart;
 8:    };
 9:
10:    struct TryCathOperationDescription
11:    {
12:      Identifier msg;
13:      any        tryDescription;
14:      any        catchDescription;
15:    };
16:  };
```

FIG. A.8 – *Interface de description d'un comportement réactif de traitement d'exceptions*

A.2 Interface de description des règles d'interaction

InteractingRule est une interface qui permet de manipuler les méta-informations associées à la description d'une règle d'interaction et donc au comportement réactif défini dans celle-ci. L'interface InteractingRule est présentée par la figure A.9 page ci-contre.

L'attribut msg décrit le message formel qui, lorsque le message qu'il contient est envoyé, déclenche le comportement réactif défini dans l'interaction et décrit par l'attribut behavior. L'attribut vars contient la liste des variables utilisées par le comportement réactif de la règle d'interaction.

Les fonctions create* permettent de construire les différents objets de dépôt décrivant les comportements réactifs. Ces fonctions vérifient que la construction du comportement réactif est cohérente et respecte la syntaxe abstraite du langage ISL. Si ce n'est pas le cas les fonctions create* renvoient la valeur *nil*.

A.3 Interface de description des schémas d'interactions

InteractionScheme est une interface qui fournit un accès aux règles d'interaction (et donc aux comportements réactifs) définies dans un schéma d'interactions et permet de manipuler les méta-informations associées à la description d'un schéma d'interactions. L'interface InteractionScheme est présentée par la figure A.10 page 184.

L'attribut class définit l'identificateur (le type) de la classe du schéma d'interactions qui est décrit par l'objet de dépôt instance de InteractionScheme.

Les deux fonctions rulesContents et allRulesContents renvoient la liste des règles d'interaction contenues dans le schéma d'interactions. Elle sont équivalentes à l'appel de la fonction contents avec la valeur isrInteratingRule pour le paramètre limitKind et TRUE pour le paramètre exclureInherited dans le cas de rulesContents, FALSE dans le cas de allRulesContents.

Les deux fonctions ruleDescribeContents et allRuleDescribeContents sont la restriction de la fonction Container::describeContents sur les objets de dépôt de type isrInteractingRule avec TRUE comme valeur pour le paramètre excludeInherited dans le cas de la fonction ruleDescribeContents, et FALSE dans le cas de la fonction allRuleDescribeContents.

La fonction getParticipants renvoie la liste des participants du schéma d'interactions. Cette liste peut être modifiée par les fonctions addParticipant, addParticipants et removeParticipant.

Les deux fonctions addParticipant et addParticipants permettent d'ajouter à cette liste un ou un ensemble de participants. Si lors d'un appel de l'une de ces deux fonctions, l'un des participants à ajouter est déjà présents dans la liste des participants du schéma d'interactions alors la fonction déclenchera l'exception CANNOT_PROCEED avec nameAlreadyExist comme raison.

```
 1:  module CosInteraction
 2:  {
 3:    struct Variable {
 4:      Identifier     name;
 5:      TypeIdentifier type;
 6:    };
 7:    typedef sequence <Variable> VariableSeq;
 8:
 9:    struct FormalMessage {
10:      Message           msg;
11:      ParDescriptionSeq params;
12:    };
13:
14:    interface InteractingRule : Contained
15:    {
16:      // read/write interface
17:      FormalMessage msg;
18:      VariableSeq  vars;
19:      Operation    behavior;
20:
21:      // write interface
22:      CallOperation createCallOperation (in Message,
                                            in ParDescriptionSeq);
23:      AssignOperation createAssignOperation (in MsgAssignOperation msg,
                                                in Identifier var);
24:      WaitOperation createWaitOperation (in CallOperation waitMsg,
                                            in MsgAssignWaitOperation msg);
25:      SequenceOperation createSequenceOperation (in Operation left,
                                                    in Operation right);
26:      ConcurrencyOperation createConcurrencyOperation (in Operation left,
                                                          in Operation right);
27:      ConditionalOperation createConditionalOperation (in CallOperation cond,
                                                          in Operation then,
                                                          in Operation else);
28:      TryCatchOperation createTryCatchOperation (in Identifier except,
                                                    in Operation try,
                                                    in Operation catch);
29:      ExceptionOperation createExceptionOperation (in Identifier except);
30:    };
31:  };
```

FIG. A.9 – *Interface de description des règles d'interaction*

La fonction `removeParticipant` permet, quant à elle, de supprimer l'un des participants du schéma d'interactions. Si une, ou plusieurs règles d'interactions, du schéma d'interactions comportent une « référence » au participant à enlever (par exemple lorsque le message déclencheur d'un comportement réactif appartient au participant qui doit être enlevé).

La fonction `createRule` permet de créer une règle d'interaction dont le message déclencheur est `msg` et dont le comportement réactif est vide (il pourra être rempli grâce aux fonctions de l'interface `InteractingRule`). Si une règle d'interaction est déjà associée à ce message déclencheur alors l'exception `CANNOT_PROCEED` est déclenchée avec `nameAlreadyExist` comme raison. Si le message déclencheur n'appartient à aucun des participants du schéma d'interactions alors l'exception `CANNOT_PROCEED` est déclenchée avec `badType` comme raison.

Enfin, la fonction `instanciate` permet de créer une instance du schéma d'interactions dont les objets participants sont définis par le paramètre `participants`. Elle renvoie l'interaction elle-même. Elle déclenche l'exception `CANNOT_PROCEED` si au moins l'un des participants n'est pas du bon type (par rapport à ceux précisés lors de la définition du schéma d'interactions).

L'appel de la fonction `describe`, héritée de l'interface `Contained`, sur un objet de dépôt représentant un schéma d'interactions retourne un objet de type `SchemeDescription` dans le champ `value` de la structure `Description`.

```
 1: module CosInteraction
 2: {
 3:   struct Participant
 4:   {
 5:     Identifier objectType;
 6:     Identifier objectName;
 7:   };
 8:   typedef sequence <Participant> ParticipantSeq;
 9:
10:   typedef string ObjectReference;
11:   typedef sequence <ObjectReference> ObjectReferenceSeq;
12:
13:   interface InteractionScheme : Container, Contained
14:   {
15:     // read/write interface
16:     Identifier class;
17:
18:     // read interface
19:     ContainedSeq rulesContents ();
20:     DescriptionSeq rulesDescribeContents ();
21:
22:     ContainedSeq allRulesContents ();
23:     DescriptionSeq allRulesDescribeContents ();
24:
25:     ParticipantSeq getParticipants ();
26:
27:     // write interface
28:     void addParticipant (in Participant part) raises (CANNOT_PROCEED);
29:     void addParticipants (in ParticipantSeq parts) raises (CANNOT_PROCEED);
30:
31:     InteractingRule createRule (in FormalMessage msg) raises (CANNOT_PROCEED);
32:
33:     CORBA::Object instanciate (in ObjectReferenceSeq participants) raises (CANNOT_PROCEED);
34:   };
35:
36:   struct SchemeDescription
37:   {
38:     Identifier name;
39:     Identifier class;
40:     Login      owner;
41:   };
42: };
```

FIG. A.10 – *Interface de description des schémas d'interactions*

Annexe B

Interface IDL du métaobjet

Nous présentons, dans cette annexe, l'interface IDL du métaobjet associé à chacun des objets du système permettant d'enregistrer les comportements réactifs déclenchés par l'un des messages de l'objet contrôlé par le métaobjet. En effet, comme cela a été vu dans le paragraphe 8.4.2, l'activation d'une interaction implique notamment, pour chacune de ses règles d'interaction, l'enregistrement de celle-ci dans le métaobjet de l'objet déclencheur de la règle d'interaction.

Cet enregistrement consiste à créer, dans le métaobjet, une instance du comportement réactif afin d'éviter, notamment, des accès distants au dépôt de schémas d'interactions et à l'instance de l'interaction lors de l'exécution de comportements réactifs ou lors de l'application de la fonction de fusion comportementale sur ces derniers. La création, sur un métaobjet, d'un comportement réactif consiste à construire l'arborescence des opérateurs réactifs (c.f. paragraphe 8.2.1) en utilisant les propriétés de la notation postfixe (utilisation du principe de pile). La figure B.1 présente l'interface IDL du métaobjet dédiée à l'enregistrement des comportements réactifs.

```
 1:  module CosInteractionMetaObject
 2:  {
 3:    typedef sequence <long> parametersSeq
 4:
 5:    interface MetaObject
 6:    {
 7:      // registration interface
 8:      void registerRule (in long ruleID, in string messageName);
 9:
10:      //write interface
11:      long createRule ();
12:      void createIdentifiers (in long ruleID, in long number);
13:
14:      // rule construction interface
15:      void insertMessage (in long ruleID, in string objectName, in string messageName,
                             in parametersSeq parameters);
16:      void insertIdentifier (in long ruleID, in long identifierNumber);
17:      void insertSequenceOperation (in long ruleID);
18:      void insertConcurrencyOperation (in long ruleID);
19:      void insertAssignOperation (in long ruleID);
20:      void insertWaitOperation (in long ruleID);
21:      void insertDelegateOperation (in long ruleID);
22:      void insertConditionalOperation (in long ruleID, in boolean elsePart);
23:      void insertExceptionOperation (in long ruleID, in string exceptName);
24:      void insertTryCatchOperation (in long ruleID, in string exceptName);
25:    };
26:  };
```

FIG. B.1 – *Interface IDL du métaobjet permettant l'enregistrement des comportemetns réactifs*

La méthode `createRule` crée une pile vide qui serira à la construction du comportement réactif à enregistrer tandis que la méthode `createIdentifiers` permet d'associer au comportement réactif des variables (les paramètres du message déclencheur sont considérés ici comme des variables).

Les méthodes `insert*` construisent un comportement réactif sur la pile en utilisant (en fonction de l'arité de l'opérateur réactif à insérer) 1, 2 ou 3 éléments en sommet de pile comme arguments.

La méthode `registerRule` enregistre le comportement réactif contenu au sommet de la pile d'enregistrement dans le métaobjet.

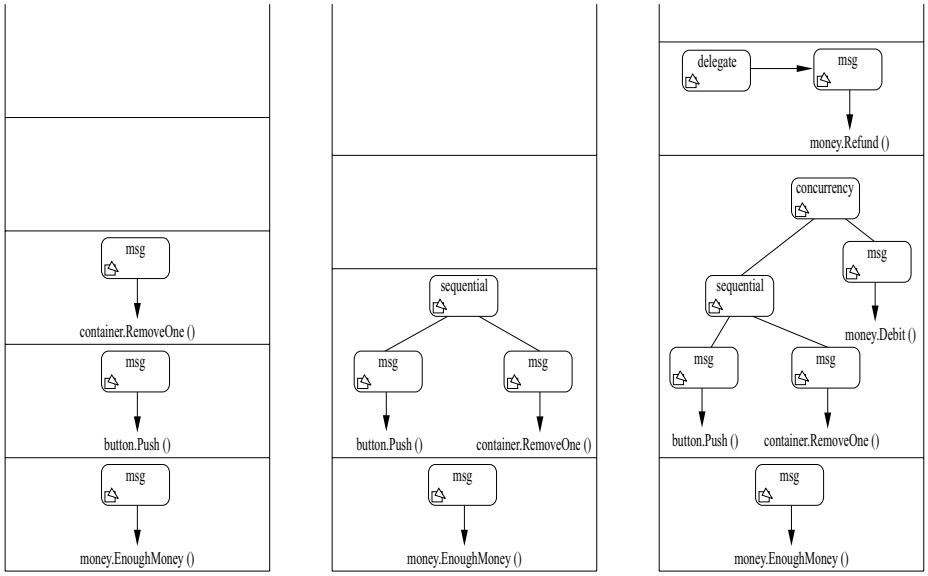
EXEMPLE. — Soit la règle d'interaction suivante dont le comportement réactif doit être enregistré auprès du métaobjet associé au message déclencheur :

```
push.Push () -> if money.EnoughMoney () then [ button.Push () ; container.RemoveOne () ] // money.Debit ()
                                        else delegate money.Refund () endif
```

Son enregistrement implique les étapes suivantes :

- Création d'une pile vide pour la construction du comportement réactif (méthode`createRule`),
- Mise sur la pile nouvellement créée des messages `money.EnoughMoney ()`, `button.Push ()` et `container.RemoveOne ()` (le contenu de la pile une fois cette étape réalisée est présenté par la figure B.2a),
- Application de la méthode `insertSequenceOperation` permettant la construction d'un comportement réactif séquentiel (le contenu de la pile une fois cette étape réalisée est présenté par la figure B.2b),
- Mise en pile du message `money.Debit ()` puis application de la méthode `insertConcurrencyOperation` permettant la construction d'un comportement réactif concurrentiel,
- Mise en pile du message `money.Refund ()` puis application de la méthode `insertDelegateOperation` permettant la construction d'un comportement réatif de délégation (le contenu de la pile une fois cette étape réalisée est présenté par la figure B.2c),
- Application de la méthode `insertConditionalOperation` permettant la construction d'un comportement réactif de condition avec partie `else`.

Le contenu de la pile une fois toutes ces étapes réalisées est le comportement réactif enregistré dans le métaobjet (par application de la méthode `registerRule`).



B.2a – B.2b – B.2c –

FIG. B.2 – *Contenu de la pile d'enregistrement d'un comportement réactif à différentes étapes*

Annexe C

Prise en compte du critère C3.2 : une solution partielle

La mise en œuvre du modèle à interactions distribuées existante lors du développement du système à base de connaissances (chapitre 10) ne supportait pas complètement le critère C3.2 (se reporter au paragraphe 9.5.2). En fait, il n'était possible de faire interagir entre eux que des objets non prévus pour être distribués ou que des objets distribués, mais pas un panachage des deux types d'objets.

Lors du développement de l'application présentée dans le chapitre 10, les objets de la base de connaissances (qui sont des objets non prévus pour être distribués) doivent pouvoir participer à des interactions dont certains des participants sont des objets des interfaces graphiques (eux aussi non prévus pour être distribués). Or ces deux types d'objets (objets de la base de connaissances et objets des interfaces graphiques) sont définis sur des sites distants. Nous avons donc conçu un mécanisme permettant à des objets non prévus pour être distribués de communiquer entre eux, via des interactions, même s'ils sont distants les uns des autres.

C.1 Permettre l'exécution à distance sur un objet non distribué

Un objet non prévu pour être distribué n'a pas été préparé pour pouvoir recevoir des messages d'un objet distant. Il n'a notamment aucune connaissance sur la manière de répondre à de tels messages distants et ne dispose d'aucune interface IDL lui permettant d'accéder au système distribué CORBA. Il est donc nécessaire de lui adjoindre une telle interface afin qu'il soit accessible à distance et puisse répondre à un objet distant. Ceci peut être réalisé grâce au schéma de conception (*design pattern*) Adaptateur (figure C.1).

Pour ce faire, une interface CORBA permettant d'encapsuler un objet non prévus pour être distribué a été définie. Cette interface, qui joue le rôle de l'objet adaptateur, est un objet distribué et est donc accessible à distance.

Bien que générer un objet d'adaptation (et donc une interface IDL) pour chaque objet non prévu pour être distribué est une solution envisageable, elle implique que chaque objet de l'application non prévu pour être distribué est associé à un objet distribué le rendant accessible à distance. Cette solution a donc un coût en mémoire non négligeable.

La solution retenue est toute autre. Elle se base sur le principe du *Dynamic Invocation Interface* (DII) de CORBA. En effet, au lieu que l'objet d'adaptation dispose d'une interface identique à celle de l'objet non prévu pour être distribuée auquel elle est associée, l'objet d'adaptation ne disposera que d'une unique méthode permettant d'appeler n'importe quelle méthode (publique) d'un objet non distribué situé sur le même site que lui. Cette solution implique, comme pour le mécanisme DII, de construire une requête et de remplir cette dernière avec les caractéristiques du message à appeler (notamment ses paramètres).

Ainsi, grâce à cette solution, il n'est plus nécessaire d'associer un objet d'adaptation par objet non prévu pour être distribué, un seul objet d'adaptation par site suffit. Mais cet objet d'adaptation implique une évaluation dynamique des messages envoyés, mécanisme qui est, comme cela a été vu dans le chapitre précédent, coûteux en temps d'exécution.

Cependant, ce mécanisme permettant de rendre accessible à distance un objet non prévu pour être distribué a été mis en place pour permettre l'instanciation d'interactions sur des objets non distribués mais situés sur des sites distants les uns des autres. C'est donc le mécanisme d'exécution des comportements réactifs qui va utiliser la solution proposée ici.

Or ce mécanisme implique, lui aussi, une évaluation dynamique des messages. Donc la solution proposée ici « décharge » le mécanisme d'exécution des comportements réactifs de l'évaluation dynamique des messages pour les objets non prévu pour être distribués mais distants du site sur lequel se déroule l'exécution du comportement réactif associé à la règle d'interaction déclenchée.

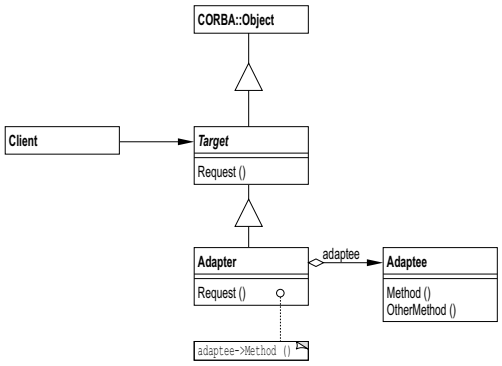


FIG. C.1 – *Rendre accessible à distance un objet non prévu pour être distribué : utilisation du schéma de conception Adaptateur*

C.2 Référencement des objets non distribués

Le mécanisme défini ci-dessus permet d'exécuter une méthode sur un objet non prévu pour être distribué. Cependant, il nécessite de disposer d'une référence sur l'objet destinataire du message. En fait, les objets non prévus pour être distribués deviennent, grâce au mécanisme ci-dessus, accessibles à distance et doivent donc disposer, comme tous les objets distribués, d'une référence permettant de les identifier de manière unique à travers tout le réseau. Il faut donc étendre le mécanisme de référencement de CORBA aux objets non distribués. Cela revient à donner à chaque objet non distribué une IOR.

Cependant, il n'est pas possible de donner aux objets non distribués l'IOR de l'objet qui va les encapsuler et les rendre accessible à distance (c'est-à-dire l'IOR de l'objet d'adaptation). En effet la sémantique de l'exécution d'une méthode sur un objet distribué est différente de celle sur un objet non distribué puisque, dans ce dernier cas, il est nécessaire de construire une requête qui sera dynamiquement évaluée au lieu d'effectuer un appel distant.

Par contre, CORBA propose plusieurs schémas d'identification des objets CORBA sous la forme de Localisateur Uniforme des Ressources (*Uniform Resource Locator*, URL). Or de nouveaux schémas d'identification peuvent être définis et intégrés à CORBA.

Ainsi il est possible de définir un nouveau schéma d'identification CORBA permettant de désigner les objets non distribués. Dans ce cas, la résolution d'un nom d'objet utilisant un tel schéma d'identification renverra l'objet CORBA qui sert d'adaptateur à l'objet non distribué. Cette solution implique cependant de modifier l'ORB de CORBA.

Une solution intermédiaire a été adoptée. Un nouveau schéma d'identification a été défini pour désigner les objets non distribués. Cependant, ce schéma n'a pas été intégré à l'ORB. Seule une interface permettant de transformer un identificateur de ce style en son objet CORBA l'encapsulant et permettant l'opération inverse a été définie.

Avec ce nouveau schéma d'identification, le nom d'un objet est une chaîne de caractères commençant par "LOR:" (pour *Local Object Reference*), suivi par le nom CORBA, au format IOR, de l'objet d'adaptation auquel sont associés les objets non distribués du site sur lequel se trouve cet objet d'adaptation, puis par une suite de 16 caractères contenant l'identifiant de l'objet non distribué dans son application (cet identifiant local est intégralement géré par l'objet d'adaptation).

C.3 Limitations de la solution

L'appel d'une méthode sur un objet distant non prévu pour être distribué implique de transmettre à l'adaptateur les paramètres du message à exécuter. Ceci implique que ces derniers transitent par le bus réseau de CORBA. Or seuls les types de base (entiers, chaînes de caractères, flottants, etc.) et les

objets distribués savent être traités par le bus réseau. De plus, pour tout autre type de données, seules les personnes utilisant ces données sont en mesure de définir comment elles doivent être transmises par le bus réseau. Afin d'être adapatable, notre solution permet la définition d'opérateurs d'insertion et d'extraction de types complexes du/vers le bus logiciel.

Ainsi cette solution, et toute autre solution basée sur le même principe, limite le type des paramètres à ceux de base et aux objets distribués. Par conséquent, seules les données dont le type est un type de base ou un objet distribué pourront être passées comme paramètre à un message distant sur un objet non prévu pour être distribué. Cette limitation n'est pas gênante dans le cadre de l'application développée pour le projet COLOR car les valeurs des attributs étaient soit des entiers, soit des chaînes de caractères.

De plus, les exceptions déclenchées par une méthode de l'objet non prévu pour être distribué ne peuvent pas être propagée à travers le bus réseau. En effet, le type de l'exception déclenchée peut ne pas exister sur le site de l'appelant. Par contre, le nom du type de l'exception peut être retournée à l'appelant en étant encapsulé par une exception définie au sein du mécanisme permettant de rendre accessible à distance un objet non distribué.

C.4 Interface CORBA

Le module de l'interface de l'adaptateur rendant accessible à distance les objets non prévus pour être distribués et du schéma d'identification est présenté par la figure C.2. Il définit un ensemble d'exceptions, un ensemble de fonctions permettant de traduire une référence LOR en son objet d'adaptation et, inversement, un objet local en sa référence LOR, ainsi que l'interface de l'adaptateur.

L'interface `RemoteExecution` définit deux fonctions. La première, `ExecuteMessage`, permet d'exécuter une méthode sur un objet non prévu pour être distribué référencé par un LOR. Les paramètres de la méthode sont « encapsulés » sous la forme d'un `any`.

Cette fonction déclenche un exception `INVALID_LOR` si la référence de l'objet est invalide (par exemple parce que l'objet ne se trouve pas sur le même site que l'objet d'adaptation). Si l'un des paramètres n'est pas du type attendu par la méthode, l'exception `BAD_PARAMETER_TYPE` est déclenchée et l'attribut `parameterPos` de l'exception indique quelle est la position du paramètre dans la méthode dont le type est incorrect.

Si la méthode à exécuter n'existe pas sur l'objet non distribué alors l'exception `CANNOT_PROCEED` est déclenchée avec `invalidMethodName` comme raison. L'exception `CANNOT_PROCEED` est déclenchée avec `invalidObject` comme raison si l'objet sur lequel la méthode doit être exécutée n'existe plus.

La seconde fonction, `object_to_string`, permet de déterminer la référence LOR d'un objet non prévu pour être distribué dont le nom est passé en paramètre. Ce nom est celui défini par le métaobjet des objets interagissants.

La méthode `string_to_object` permet d'obtenir l'objet distribué d'adaptation (interface `RemoteExecution`) d'un objet non prévu pour être distribué dont la référence est fournie en paramètre.

```
 1: module CosInteraction
 2: {
 3:   module RemoteAccess
 4:   {
 5:     typedef string LOR;
 6:
 7:     exception BAD_PARAMETER_TYPE
 8:     {
 9:       long parameterPos;
10:     }
11:
12:     enum CannotProceedReason
13:     {
14:       invalidMethodName, invalidObjectReference, others
15:     };
16:
17:     exception CANNOT_PROCEED
18:     {
19:       CannotProceedReason reason;
20:     };
21:
22:     enum InvalidLORReason
23:     {
24:       invalidFormat, invalidLOR, invalidObject
25:     };
26:
27:     exception INVALID_LOR
28:     {
29:       InvalidLOR why;
30:     };
31:
32:     interface RemoteExecution
33:     {
34:       any ExecuteMessage (in LOR, in string methodName, in any parameter)
                          raises (INVALID_LOR, CANNOT_PROCEED, BAD_PARAMETER_TYPE);
35:       LOR object_to_string (in string objectName);
36:     };
37:
38:     interface RemoteExecutionManager
39:     {
40:       RemoteExecution string_to_object (in LOR object) raises (INVALID_LOR);
41:     };
42:   };
43: };
```

FIG. C.2 – *Interface* CORBA *permettant de rendre accessible à distance un objet non distribué*

Références Bibliographiques

[AB98] M. AKSIT et L. BERGMANS. « Examples of Reusing Synchronization Code in Aspect-Oriented Programming using Composition-Filters ». Dans *Proceedings of the 5th. Maghrebian Conference on Software Engineering and Artificial Intelligence (MCSEAI'98)*, pages 257–272, Tunis (Tunisie), Décembre 1998.

[ABC+96] Y. AMIR, D. BREITGAND, G. CHOCKLER et D. DOLEV. « Group Communication as an Infrastructure for Distributed System Management ». Dans *International Workshop on Services in Distributed and Networked Environment*, pages 84–91, Juin 1996.

[ABV92] M. AKSIT, L. BERGMANS et S. VURAL. « An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach ». Dans *Proceedings of ECOOP'92*, LNCS 615, pages 372–395, Berlin (Allemagne), Juin 1992. Springer-Verlag.

[ADG98] R. ALLEN, R. DOUENCE et D. GARLAN. « Specifying and Analyzing Dynamic Software Architectures ». Dans *Proceedings of 1998 Conference on Fundamental Approaches to Software Engineering*, LNCS 1382, pages 21–37, Lisbon (Portugal), 1998. Springer-Verlag.

[ADH+98] H. ABELSON, R.K. DYBVIG, C.T. HAYNES, G.J. ROZAS, N.I. Adams IV, D.P. FRIEDMAN, E. KOHLBECKER, G.L. Steele JR., D.H. BARTLEY, R. HALSTEAD, D. OXLEY, G.J. SUSSMAN, G. BROOKS, C. HANSON, K.M. ITMAN et M. WAND. « Revised ^{5} Report on the Algorithmic Language Scheme ». *Higher-Order and Symbolic Computation*, volume 11, numéro 1, pages 7–105, Août 1998.

[AFP+93] G. AGHA, S. FRØLUND, R. PANWAR et D. STURMAN. « A Linguistic Framework for Dynamic Composition of Dependability Protocols ». Dans *Proc of DCCA-3*, pages 197–207, 1993.

[AG94] R. ALLEN et D. GARLAN. « Formal Connectors ». Rapport Technique CMU-CS-94-115, School of Computer Science, Carnegie Mellon University, Pittsburgh (USA), Mars 1994.

[AG97] R. ALLEN et D. GARLAN. « A Formal Basis for Architectural Connection ». Dans *ACM Transaction on Software Engineering and Methodology (TOSEM)*, volume 6, numéro 3, pages 213–249, Juillet 1997.

[Agh97] G. AGHA. « Abstracting Interaction Patterns: A Programming Paradigm for Open Distributed Systems ». Dans *Fomal Methods for Open Object-based Distributed Systems*. 1997, IFIP Transactions.

[All97] R.J. ALLEN. « *A Formal Approach to Software Architecture* ». Thèse de Doctorat, Carnegie Mellon University, School of Computer Science, Mai 1997. available as TR# CMU-CS-97-144.

[ALP99] Uwe ASSMANN, A. LUDWIG et D. PFEIFER. « Programming Connectors In an Open Language ». Dans *Proceedings of Position Papers, WICSA 1, Worksing IFIP Conference on Software Architecture, IFIP WG 2.9*, Février 1999.

[ALZ00] D. ANCONA, G. LAGORIO et E. ZUCCA. « Jam - A Smooth Extension of Java with Mixins ». Dans E. BERTINO, éditeur, *Proceedings of 14th European Conference on Object-Oriented Programming (ECOOP'2000)*, LNCS 1850, pages 154–178, Cannes et Sophia Antipolis (France), Juin 2000. Springer Verlag.

[AOP01] « Overview of Aspect-Oriented Programming », Mars 2001. `http://www.parc.xerox.com/csl/projects/aop/overview.shtml`.

[AP97] A. AMANDI et A. PRICE. « Object-oriented agent programming through the brainstorm system ». Dans *Proceedings of Practical Applications of Intelligent Agents and Multi-Agents (PAAM'97)*, Londres (Royaume-Uni), Avril 1997.

[AP98] A. AMANDI et A. PRICE. « Building object-agents from a software meta-architecture ». Dans Flávio Moreira de OLIVEIRA, éditeur, *Advances in Artificial Intelligence*, 14^{th} *Brazilian Symposium on Artificial Intelligence (SBIA'98)*, LNCS 1515, pages 21–30, Porto Alegre (Brésil), Novembre 1998. Springer-Verlag.

[AT98] M. AKSIT et B. TEKINERDOGAN. « Aspect-Oriented Programming Using Composition Filters ». Dans *Position paper for the Aspect Oriented Programming workshop at the European Conference on Object-Oriented Programming (ECOOP'98)*, LNCS 1543, page 435, Bruxelles (Belgique), Juillet 1998. Springer-Verlag.

[AW+93] M. AKSIT, K. WAKITA et al. « Abstracting object interactions using composition filters ». Dans *Proceedings of ECOOP'93 Workshop on Object-Based Distributed Programming*, pages 152–184, Kaiserslautern (Allemagne), Juillet 1993. Springer-Verlag.

[AZI99] A. AMANDI, A. ZUNINO et R. ITURREGUI. « Multi-paradigm languages supporting multi-agent development ». Dans F.J. GARIJO et M. BOMAN, éditeurs, *Multi-Agent System Engineering*, 9^{th} *European Workshop on Modeling Autonomous Agents in a Multi-Agent World, MAAMAW'99*, LNCS 1647, pages 128–139, Valencia (Espagne), Juin 1999. Springer-Verlag.

[BAB+96] L. BELLISSARD, S. B. ATALLAH, F. BOYER et M. RIVEILL. « Distributed Application Configuration ». Dans *Proceedings od 16th International Conference on Distributed Computing Systems (ICDCS'96)*, pages 579–585, Hong Kong, Mai 1996.

[BCJ+95] S. BAEG, J. CHOI, M. JANG, S. PRK, B. MIN et Y. LIM. « The Integration of Heterogeneous Applications using Plug-and-Play ». Dans *Transactions of the Korea Information Processing Society*, volume 2, numéro 6, pages 947–959, Hong-Kong, Novembre 1995. IEEE Computer Society.

[BCR+98] G. BLAIR, G. COULSON, P. ROBIN et M. PAPATHOMAS. « An Architecture for Next Generation Middleware ». Dans *Middleware'98*, The Lake District (Royaume Uni), Novembre 1998.

[BDF98] L. BERGER, A-M. DERY et M. FORNARINO. « Interactions Between Objects: an Aspect of Object-Oriented Languages ». Dans *ICSE'98 Workshop on Aspect-Oriented Programming*, Kyoto (Japon), Avril 1998.

[BDG+88] D. BOBROW, L. DEMICHIEL, R. GABRIEL, S. E. KEENE, G. KICZALES et D. A. MOON. « Common Lisp Object System Specification ». Dans *SIGPLAN Notices*, volume 23 (Special Issue). Septembre 1988, ACM Press.

[BEL+95] M. BLAHA, W.P.F. EDDY, W. LORENSEN et J. RUMBAUGH. *Modélisation et conception orientées objet*. Masson - Prentice Hall, 2nd édition, 1995.

[Ber94] L. BERGMANS. « *Composing Concurrent Object* ». Thèse de Doctorat, University of Twente, Enschede (Pays Bas), 1994.

[Ber98] L. BERGER. « Compile Time and Runtime Reflection for Dynamic Evaluation of Messages: Application to Interactions between Remote Objects ». Dans *OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, SIGPLAN Notices, volume 23, Vancouver (Canada), Octobre 1998. ACM Press.

[Ber99] L. BERGER. « Évaluation Dynamique des Messages dans un Environnement Compilé et Distribué : Application aux Interactions entre Objets Distants ». Dans *Actes du Colloque Langages et Modèles à Objets*, pages 131–146, Villefranche-sur-Mer (France), Janvier 1999. Hermès Science. ISBN 2-7462-0008-2.

[Ber00] L. BERGER. « Junction Point Aspect: A Solution to Simplify Implementation of Aspect Languages and Dynamic Management of Aspect Programs ». Dans *ECOOP'2000 Poster*, Cannes et Sophia Antipolis (France), Juin 2000.

[Ber01] L. BERGER. « Interactions et modèles de programmation : Support des interactions par les modèles à objets et à composants ». *Journal l'Objet*, 2001. À paraître.

[BF95] M. BARBUCEANU et M.S. FOX. « COOL: A Language for Describing Coordination in Multi Agent Systems ». Dans *Proceedings of First International Conference on Multi Agent Systems (ICMAS-95)*, pages 17–24, San Francisco (Californie, USA), Juin 1995. AAAI Press.

[BFJ+] J. BRANT, B. FOOTE, R.E. JOHNSON et D. ROBERTS. « Wrappers to the rescues ». Springer-Verlag.

[BG96] J.-P. BRIOT et R. GUERRAOUI. « Smalltalk for Concurrent and Distributed Programming ». *Informatik/Informatique Journal, Swiss Informaticians Society, numéro 1*, pages 16–19, Février 1996.

[BGW93] D.G. BOBROW, R.P. GABRIEL et J.L. WHITE. « *Object-Oriented Programming: the* CLOS *Perspectives* », Chapitre CLOS in Context, pages 29–61. The MIT Press, Cambridge (Massachusetts, USA), 1993.

[Bir93] K.P. BIRMAN. « The process group approach to reliable distributed computing ». Dans *Communication of the ACM*, SIGPLAN Notices, volume 36, numéro 12, pages 39–53. Décembre 1993, ACM Press.

[BJR97] G. BOOCH, I. JACOBSON et J. RUMBAUGH. « Unified Modeling Language (UML) », Janvier 1997. *Rational Software Corporation*, version 1.0.

[BKdR91] D. G. BOBROW, G. KICZALES et J. des RIVIERES. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[BL91] J. Van Den BOS et C. LAFFRA. « PROCOL, A Concurrent Object-Oriented Language with Protocols Delegation and Constraints ». *Acta Informatica*, volume 28, pages 511–538, 1991.

[BN84] A.D. BIRRELL et B.J. NELSON. « Implementing Remote Procedure Calls ». Dans *ACM Transactions on Computer Systems*, TOCS, volume 2, numéro 1, pages 39–59, 1984.

[Bos95a] J. BOSCH. « *The layered object model* ». Thèse de Doctorat, University of Twente, Enschede (Pays Bas), 1995.

[Bos95b] J. BOSCH. « *Layered Object Model: Investigating Paradigm Extensibility* ». Thèse de Doctorat, Lund University, Department of Computer Science, Octobre 1995.

[Bos95c] J. BOSCH. « Relations as Object Model Components ». Dans *Journal of Programming Languages*, 1995.

[Bou91] F. BOUSSINOT. « Reactive C: An Extension of C to Program Reactive Systems ». Dans *Software Practice and Experience*, volume 21, numéro 4, pages 401–428, 1991.

[BPC+95] S. BAEG, S. PARK, J. CHOI, M. JANG et Y. LIM. « Cooperation in Multi-agent Systems ». Dans *Intelligent Computer Communications (ICC495)*, pages 1–12, Cluj-Napoca (Roumanie), Juin 1995.

[BR89] T. BIGGERSTAFF et C. RICHTER. « *Reusability framework, assessment, and directions* », Chapitre 1, pages 1–17. Frontier Series: Software reusability: Volume I - Concepts and Models. ACM Press, New York (USA), 1989.

[BR96] Luc BELLISSARD et Michel RIVEILL. « From Distributed Objects to Distributed Components: The OLAN Approach ». Dans *Workshop Putting Distributed Objects to Work*, LNCS 1098, Linz (Autriche), Juillet 1996. ECOOP'96, Springer-Verlag.

[BR99] E. BRUNETON et M. RIVEILL. « JavaPod : un bus logiciel adaptable et extensible ». Rapport Technique, INRIA, Novembre 1999.

[Bra96] J. BRANT. « Method Wrappers. SmallTalk Goodies », 1996. `http://st-www.cs.uiuc.edu/users/brant/Applications/MethodWrappers.html`.

[Bri89] J.P. BRIOT. « Actalk: a Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment ». Dans *Proceedings of European Conference on Object-Oriented Programming (ECOOP'89)*. 1989, Springer-Verlag.

[Bri99] J-P. BRIOT. « Actalk: A Framework for Object-Oriented Concurrent Programming - Design and Experience ». Dans *Object-Based Parallel and Distributed Computing II - Proceedings of the 2nd France-Japan Workshop (OBPDC'97)*. 1999, Hermès Science.

[Bro94] K. BROCKSCHMIDT. *Inside OLE 2*. Microsoft Press, 1994.

[Bry94] D. BRYAN. « Using Rapide to model and specify inter-object behavior ». Dans *OOPSLA'94 Workshop on Precise Behavioral Specifications in OO Information Modeling*, SIGPLAN Notices, volume 29, numéro 10. Octobre 1994, ACM Press.

[BSS91] K. BIRMAN, A. SCHIPEZR et P. STEPHENSON. « Lightweight Causal and Atomic Group Multicast ». Dans *ACM Transactions On Computer Systems*, volume 9, numéro 3, pages 272–314, Août 1991.

[Car95] F.M. CARRANO. *Data Abstraction and Problem Solving with C++*. The Benjamin/Cummings Publishing Company Inc., Redwood City (Californie, USA), 1995.

[CCK98] G.A. COHEN, J.S. CHASE et D.L. KAMINSKY. « Automatic Program Transformation with JOIE ». Dans *USENIX Annual Technical Conference '98*, 1998.

[CCW+94] P. COHEN, A. CHEYER, M. WANG et S. BAEG. « An Open Agent Architecture ». Dans *Working Notes of AAAI Spring Symposium on Software Agents*, pages 1–8, 1994.

[CD99] J.C. CRUZ et S. DUCASSE. « A Group Based Approach for Coordinating Active Objects ». Dans *Proceedings of* COORDINATION'99, LNCS 1594, pages 355–371, Amsterdam (Pays Bas), Avril 1999. Springer-Verlag.

[CFM98] P. CIANCARINI, F. FRANZÉ et C. MASCOLO. « A Coordination Model to Specify Systems including Mobile Agents ». Dans *Proceedings of 9th ACM-IEEEE International Workshop on Software Specification and Design (IWSSD)*, Japan, 1998.

[CFM00] P. CIANCARINI, F. FRANZÈ et C. MASCOLO. « Using a Coordination Language to Specify and Analyse Systems Containing Mobile Components ». *ACM Transactions on Software Engineering and Methodology*, volume 9, numéro 2, pages 167–198, Avril 2000.

[CH96] G. CORNELL et C. S. HORSTMANN. *Core JAVA*. SunSoft Press, 1996. ISBN 0-13-565755-5.

[Chi93] S. CHIBA. « Open C++ Release 1.2 Programmer Guide ». Rapport Technique, Department of Information Science, University of Tokyo, 1993.

[Chi95] S. CHIBA. « A Metaobject Protocol for C++ ». Dans *Proceedings of the ACM Conference on Object-Oriented Progamming Systems, Languages, and Applications (OOPSLA'95)*, SIGPLAN Notices, volume 30, numéro 10, pages 285–299, Austin (Texas, USA), Octobre 1995. ACM Press.

[Chi97] S. CHIBA. « Implementation Techniques for Efficient Reflective Languages ». Rapport Technique, Department of Information Science, University of Tokyo, 1997.

[Chi98] S. CHIBA. « Javassist – A Reflection-based Programming Wizard for Java ». Dans *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, pages 51–55, Vancouver (British Columbia, Canada), Octobre 1998.

[Chi00] S. CHIBA. « Load-time Structural Reflection in Java ». Dans *Proceedings of ECOOP'2000*, LNCS 1850, pages 313–336, Cannes et Sophia-Antipolis (France), Juin 2000. Springer Verlag.

[Cia91] P. CIANCARINI. « PoliS: a Programming Model for Multiple Tuple Spaces ». Dans C. GHEZZI et G. ROMAN, éditeurs, *Proceedings of 6th ACM/IEEE International Workshop on Software Specification and Design (IWSSD)*, pages 44–51, Como (Italie), Octobre 1991. IEEE Computer Society Press.

[CM93] S. CHIBA et T. MASUDA. « Designing an Extensible Distributed Language with a Meta-Level Architecture ». Dans *Proceedings of European Conference on Object-Oriented Programming (ECOOP'93)*, LNCS 707, pages 483–501. 1993, Springer-Verlag.

[Coi87] P. COINTE. « Metaclass are First Class: The ObjVLisp Model ». Dans *Proceedings of OOPSLA'87*, SIGPLAN Notices, volume 22, numéro 12, pages 156–167, Orlando (Floride, USA), Octobre 1987. ACM Press.

[CR97] P. CIANCARINI et D. ROSSI. « Jada: Coordination and Communication for Java Agents ». Dans J. VITEK et C. TSCHUDIN, éditeurs, *Mobile Agent Systems: Towards the Programmable Internet*, LNCS 1222, pages 213–228, Berlin (Allemagne), 1997.

[CWM01] « Data Warehousing, CWM and MOF Resource Page », 2001. http://www.omg.org/cwm/.

[Dah99] M. DAHM. « Byte Code Engineering with the JavaClass API ». Rapport Technique B-17-98, Institut für Informatik, Freie Universität Berlin, Janvier 1999.

[DBD95] S. DUCASSE, M. BLAY et A.M. DERY. « A Reflective Model for First Class Dependencies ». Dans *OOPSLA'95: 10th Object Oriented Programming Systems Languages and Applications*, SIGPLAN Notices, volume 30, numéro 10, pages 265–280, Austin (Texas, USA), Octobre 1995. ACM Press.

[Des88] T. DESPEYROUX. « TYPOL: a formalism to implement Natural Semantics ». Rapport Technique 94, INRIA, 1988.

[DGV+95] M. DODANI, B.K. GAN, L. VELASQUEZ et X. YANG. « Modeling object interactions as first class citizens ». Dans *Poster Panel at OOPSLA'95*, Austin (Texas, USA), Octobre 1995.

[DKM+84] N. DULAY, J. KRAMER, J. MAGEE, M. SLOMAN et K. TWIDLE. « *The Conic configuration language, version 1.3* ». Imperial College Research Report Doc 84/20, Novembre 1984.

[DM95] F.-N. DEMERS et J. MALENFANT. « Reflection in logic, functional and object-oriented programming: a Short Comparative Study ». Dans *Prooceedings of the IJCAI'95 workshop on Reflection and Meta-Level Architectures and their Applications in AI*, pages 29–38, 1995.

[DM96] D. DOLEV et D. MALKI. « The Transis Approach to High Availability Cluster Communication ». *Communication of the ACM*, volume 39, numéro 4, 1996.

[Dow98] T. DOWNING. *Java RMI: Remote Method Invocation*. IDG Book Worldwide, 1998.

[DR97] S. DUCASSE et T. RICHNER. « Executable Connectors: Towards Reusable Design Elements ». Dans *Proceedings of ESEC/FSE'97 (European Software Engineering Conference)*, LNCS 1301, pages 483–500. Septembre 1997, Springer-Verlag.

[DTH+98] B. DUMANT, F. Dang TRAN, F. HORN et J.-B. STEFANI. « Jonathan: an Open Distributed Processing Environnement in Java ». Dans *Middleware'98: IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, The Lake District (Royaume Uni), Septembre 1998.

[Duc97a] S. DUCASSE. « Des Techniques de Contrôle de l'Envoi de Messages en SmallTalk ». Dans *L'objet, numéro spécial SmallTalk en France : état de l'art et de la pratique*. 1997, Hermès édition.

[Duc97b] S. DUCASSE. « *Intégration Réflexive de Dépendances dans un Modèle à Classes* ». Thèse de Doctorat, Université de Nice-Sophia Antipolis, Janvier 1997.

[Dug90] P. DUGERDIL. *SmallTalk-80, Programmation par objets*. Presses Polytechniques et Universitaires romandes, collection informatique, 1990.

[EHT95] J. Van Den ELST, F. Van HERMELEN et M. THONNAT. « Modeling Software Components for Reuse ». Dans *Seventh International Conference on Software Engineering and Knowledge Engineering*, Knowledge Systems Institute, Juin 1995.

[EMS95] P.D. EZHILCHELVAN, R.A. MACEDO et S.K. SHRIVASTAVA. « Newtop: A Fault-Tolerant Group Communication Protocol ». Dans *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95)*, pages 296–306, Vancouver (British Columbia, Canada), Juin 1995. IEEE Computer Society Press.

[Eng97] R. ENGLANDER. *Developing Java Beans*. O'Reilly & Associates, Inc., 1997.

[F+92] T. FININ et al. « Specification of the KQML Agent Communication Language ». Dans *The DARPA Knowledge Sharing Initiative, External Interfaces Working Group*, 1992.

[FA93] S. FRØLUND et G. AGHA. « A Language Framework for Multi-Object Coordination ». Dans *European Conference on Object-Oriented Programming'93 (ECOOP'93)*, pages 347–360. 1993, Springer-Verlag.

[Fer89] J. FERBER. « Computational Reflection in Class Based Object Oriented Languages ». Dans *Proceedings of OOPSLA'89*, ACM Press, pages 317–326, Octobre 1989.

[Fer95] J. FERBER. *Les systèmes multi-agents : vers une intelligence collective*. InterEdition, 1995.

[FFM+94] T. FININ, R. FRITZON, D. MCKAY et R. MCENTIRE. « KQML - An Information and Knowledge Exchange Protocol ». Dans F. KAZUHIRO et Y. TOSHIO, éditeurs, *Knowledge Building and Knowledge Sharing*. 1994, Ohmsha and IOS Press.

[FGS98] P. FELBER, R. GUERRAOUI et A. SCHIPER. « The Implementation of a CORBA Object Group Service ». *Theory and Practive of Object Systems*, volume 4, numéro 2, pages 93–105, 1998.

[FGW00] P. FELBER, R. GUERRAOUI et M. WIESMANN. « Programming with Object Group in CORBA ». *IEEE Concurrency*, volume 8, numéro 1, pages 48–58, Janvier 2000.

[FIP97] « The Foundation for Intelligent Physical Agents. The FIPA'97 Specification », 1997. http://drogo.cselt.it/fipa/spec/fipa97/fipa97.htm.

[FIP98] « Agent Management », 1998. FIPA'98, version 1.0, http://www.fipa.org/spec/FIPA98.html.

[Fla97] D. FLANAGAN. *Java in a Nutshell*. O'Reilly edition, 2nd édition, 1997.

[FP90] M. FORNARINO et A.M. PINNA. « *Un modèle objet logique et relationnel. Le langage Othelo* ». Thèse de Doctorat, Université de Nice-Sophia Antipolis, INRIA Sophia Antipolis, Avril 1990.

[Gal96] E. GALLESIO. « Designing a Meta Protocol to Wrap a Standard Graphical Toolkit ». Dans *Proceedings of ISOTAS'96*, LNCS 1049, pages 135–156. 1996, Springer-Verlag.

[Gar95] D. GARLAN, éditeur. *Proceedings of First International Workshop Architectures for Software Systems*, Avril 1995.

[GBC+93] B. GLADE, P. BIRMAN, R. COOPER et R. Van RENESSE. « Light-Weight Process Groups in the ISIS System ». Dans *Distributed Systems Engineering*, Juillet 1993.

[GC96] B. GOWING et V. CAHILL. « Meta-Object Protocol for C++: The Iguana Approach ». Dans *Proceedings of Reflection'96*, San Francisco (Californie, USA), Avril 1996.

[GGM93] B. GARBINATO, R. GUERRAOUI et K. MAZOUNI. « Distributed Programming in GARF ». Dans *Object-Based Distributed Programming*, pages 225–240. 1993, Springer-Verlag.

[GGM99] J.-M. GEIB, Ch. GRANSART et Ph. MERLE. *CORBA : des concepts à la pratique*. Dunod, 2nde édition, 1999.

[GHJ+95] E. GAMMA, R. HELM, R. JOHNSON et J. VLISSIDES. *Design Patterns – Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.

[GHM+99] A. GUILLEMET, G. HAÏK, T. MEURISSE, J.-P. BRIOT et M. LHUILLIER. « Mise en œuvre d'une approche componentielle pour la conception d'agents ». Dans Marie-Pierre GLEIZES et Pierre MARCENAC, éditeurs, *Septièmes Journées Francophones sur l'Intelligence Artificielle Distribuée et les Systèmes Multi-Agents (JFIADSMA'99)*, Paris (France), Novembre 1999. Hermes Science Publications.

[GJS96] J. GOSLING, B. JOY et G. STEELE. *The Java Language Specification*. The Java Series, Addison-Wesley, Massachusetts (USA), 1996.

[GK99] M. GOLM et J. KLEINÖDER. « Jumping to the Meta Level Behavioral Reflection Can Be Fast and Flexible ». Dans *Proceedings of Meta-Level Architectures and Reflection*, LNCS 1616, pages 22–39, Saint-Malo (France), Juillet 1999. Springer Verlag.

[GPT95] D. GARLAN, F.N. PAULISCH et W.F. TICHY, éditeurs. « *ACM Software Engineering Notes* », Chapitre Summary of the Dagstuhl Workshop on Software Architecture, pages 63–83. Juillet 1995.

[GQ94] M. GORLICK et A. QUILICI. « Visual Programming in the Large versus Visual Programming in the Small ». Dans *Proceedings of IEEE Symposium on Visual Languages*, pages 137–144, Octobre 1994.

[GR83] A. GOLDBERG et D. ROBSON. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, Massachusetts (USA), 1983.

[GR91] M. GORLICK et R.R. RAZOUK. « Using Weaves for Software Construction and Analysis ». Dans *Proceedings of 13th International Conference on Software Engineering (ICSE13)*, pages 23–34, Mai 1991.

[HHD98] R. HAYTON, A. HERBERT et D. DONALDSON. « FlexiNet: A Flexible, Component Oriented Middleware System ». Dans *SIGOPS'98*, Sintra (Portugal), Septembre 1998.

[HHG90] A.R. HELM, I.M. HOLLAND et D. GANGOPADHYAY. « Contracts: Specifying Behavioral Compositions in Object-Oriented Systems ». Dans *Proceedings of OOPSLA'90*, Special Issue of SIGPLAN Notices, pages 169–180, Ottawa (Canada), Octobre 1990. ACM Press.

[Hol92] I.M. HOLLAND. « Specifying reusable components using Contracts ». Dans *Proceedings of the 1999 European Conference on Object-Oriented Programming (ECOOP'92)*, LNCS 615, pages 287–308, Utrecht (Pays Bas), Juin 1992. Springer-Verlag.

[Hor98] B.C. HORLING. « A Reusable Component Architecture for Agent Construction ». Rapport Technique 1998-49, Department of Computer Science, University of Massachusetts, Octobre 1998.

[IKM+97] D. INGALLS, T. KAEHLER, J. MALONEY, S. WALLACE et A. KAY. « Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself ». Dans *Proceedings of OOPSLA'97*, SIGPLAN Notices, Atlanta (Georgie, USA), Octobre 1997. ACM Press.

[Ing78] D. H. INGALLS. « The Smalltalk-76 Programming System Design and Implementation ». Dans *Proceedings of 5th POPL*, pages 9–17, Tuscon (Arizona, USA), 1978.

[INR89] INRIA. « *The CENTAUR User's Manual* ». Sophia-Antipolis, France, 1989.

[INR98] INRIA, éditeur. *Langage et Modèles à Objets, État des recherches et perspectives*. INRIA, 1998.

[IR01] « Common Object Request Broker Architecture (CORBA) v2.4.2, Chapitre 10: The Interface Repository », Février 2001.

[ISS00] « IDLscript Specification », Août 2000. Object-Oriented Concepts Inc., Laboratoire d'Informatique Fondamentale de Lille.

[Jau00] O. JAUTZY. « *Intégration de source de données hétérogènes : Une Approche Langage* ». Thèse de Doctorat, École Nationale des Ponts et Chaussées, Mars 2000.

[JC99] J. JANG et J. CHOI. « A Java-based Agent Management System for Dynamic Invocation of Heterogeneous Agents ». Dans *The 1999 International Conference on Artificial Intelligence*, pages 324–330, Las Vegas (Colorado, USA), 1999.

[Jeo98] H. JEON. « JATLite », 1998. `http://java.stanford.edu`.

[JSH+96] R. JUNGCLAUS, G. SAAKE, T. HARTMANN et C. SERNADAS. « Troll - A Language for Object-Oriented Specification of Information Systems ». Dans *ACM Transactions on Information Systems*, volume 14, numéro 2, pages 175–211, Avril 1996.

[KCA+96] R. KAZMAN, P. CLEMENT, G ABOWD et L. BASS. « Classifying Architectural Elements as a Foundation for Mechanism Matching ». Dans *Proc ACM SIGSOFT Symposium on the Foundation of Software Engineering*, 1996.

[KG96] J. KLEINÖDER et M. GOLM. « MetaJava: An Efficient Run-Time Meta Architecture for Java ». Dans *Proceedings of the International Workshop on Object Orientation in Operating Systems (IWOOS'96)*, IEEE Society, 1996.

[KHH+01] G. KICZALES, E. HILSDALE, J. HUGUNIN, M. KERSTEN, J. PALM et W.G. GRISWOLD. « An Overview of AspectJ ». Dans *Proceedings of ECOOP'2001*, 2001. À paraître.

[Kic92] G. KICZALES. « Towards a New Model of Abstraction in the Engineering of Software ». Dans *Proceedings of IMSA'92 Workshop on Reflection and Meta-Level Architecture*, 1992.

[Kie96] T. KIELMANN. « Designing a Coordination Model for Open Systems ». Dans *Proceedings of* COORDINATION'96, LNCS 1061, pages 267–284, 1996.

[KLM83] G. KAHN, B. LANG et B. MÉLÈSE. « Metal: a formalism to specify formalisms ». Dans *Science of Computer Programming*, volume 3, North-Holland, 1983.

[KLM+97] G. KICZALES, J. LAMPING, A MENDHEKAR, C. MAEDA, C. LOPES, J-M. LOINGTIER et J. IRWIN. « Aspect-Oriented Programming ». Dans *Proceeding of ECOOP'97*, LNCS 1241, pages 220–242, Jyväskylä (Finlande), Juin 1997. Springer Verlag.

[KM85] J. KRAMER et J. MAGEE. « Dynamic Configuration for Distributed Systems ». *IEEE Transactions on Software Engineering*, volume 11, numéro 4, pages 424–436, Avril 1985.

[KMS+84] J. KRAMER, J. MAGEE, M. SLOMAN, K. TWIDLE et N. DULAY. « *The Conic programming language, version 2.4* ». Imperial College Research Report Doc 84/19, Octobre 1984.

[KMS+92] J. KRAMER, J. MAGEE, M. SLOMAN et N. DULAY. « Configuration Object-Based Distributed Programs in REX ». *IEEE Software Engineering Journal*, volume 7, numéro 2, pages 139–149, Mars 1992.

[Knu64] D.E. KNUTH. « Backus Normal Form vs. Backus Naur Form ». *Comm. ACM*, volume 7, numéro 12, pages 735–736, 1964.

[Koo95] P. KOOPMANS. « *Sina user's guide and reference manual* ». Dept. of Computer Science, University of Twente, 1995.

[KP88] G.E. KRASNER et S.T. POPE. « A cookbook for using the Model-View-Controller User interface paradigm in Smalltalk-80 ». Dans *JOOP*, pages 26–49, Août 1988.

[Lam87] D. A. LAMB. « IDL: Sharing Intermediate Representations ». Dans *ACM Transactions on Programming Languages and Systems*, volume 9, numéro 3, pages 297–318, Juillet 1987.

[LB97] C. LUNDBERG et J. BOSCH. « Modeling Causal Connections Between Objects ». *Journal of Programming Language*, 1997.

[Led98] T. LEDOUX. « *Réflexion dans les systèmes répartis : application à* CORBA *et Smalltalk* ». Thèse de Doctorat, Département Informatique de l'École des Mines de Nantes, Nantes (France), Mars 1998.

[LF93] Y. LABROU et T. FININ. « A Semantics Approach for KQML - A General Purpose Communication Language for Software Agents », 1993.

[LF97] Y. LABROU et T. FININ. « Semantics and conversations for an agent communication language ». Dans *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, volume M. Huhns and M. Singh, Nagoya (Japon), 1997.

[LF98] Y. LABROU et T. FININ. « Semantics for an agent communication language ». Dans M. WOOLDRIDGE, J.P. MULLER et M. TAMBE, éditeurs, *Agent Theories, Architectures and Languages IV*, Lecture Notes in Artificial Intelligence. 1998, Springer-Verlag.

[LFP99] Y. LABROU, T. FININ et Y. PENG. « The current landscape of Agent Communication Languages ». Dans *Intelligent Systems*, volume 14, numéro 2. Avril 1999, IEEE Computer Science.

[Lhu98] M. LHUILLIER. « *Une approche à base de composants logiciels pour la conception d'agents. Principes et mise en œuvre à travers la plate-forme Maleva* ». Thèse de Doctorat, Université Paris VI, Février 1998.

[LK97] C. Videira LOPES et G. KICZALES. « D: A Language Framework for Distributed Programming ». Rapport Technique SPL97-010 P9710047, Xerox PARC, Février 1997.

[LK99] C. Videira LOPES et G. KICZALES. « Aspect-Oriented Programming with Aspect-J », 1999. Xerox PARC, Tutorial, http://www.aspectj.org.

[LKA+95] D.C. LUCKHAM, J.J. KENNEY, L.M. AUGUSTIN, J. VERA, D. BRYAN et W. MANN. « Specification and Analysis of System Architecture using RAPIDE ». Dans *IEEE Transactions on Software Engineering*, volume 21, numéro 9, pages 336–355, Avril 1995.

[LKR+92] J. LAMPING, G. KICZALES, L. RODRIGUEZ et E. RUF. « An Architecture for Open Compiler ». Dans *Proceedings of IMSA'92, Workshop Reflection and Meta-Level Architectures*, pages 95–106, Tokyo (Japon), Novembre 1992.

[LLM99] K. LIEBERHERR, D. LORENZ et M. MEZINI. « Programming with Aspectual Components ». Rapport Technique NU-CSS-99-01, College of Computer Science, Northeastern University, Avril 1999.

[LS97] E. LUPU et M. SLOMAN. « A Policy Based Role Object Model ». Dans *EDOC'97*, Gold Coast (Queensland, Autralie), Octobre 1997. IEEE.

[Luc96] D.C. LUCKHAM. « RAPIDE: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events ». Dans *DIMACS Partial Order Methods Workshop IV*, Princeton University, Août 1996.

[Mae87a] P. MAES. « *Computational Reflection* ». Thèse de Doctorat, Artificial Intelligence Laboratory, Vrije Universiteit, Bruxelles (Belgique), 1987.

[Mae87b] P. MAES. « Concept and Experiments in Computational Reflection ». Dans *Proceedings of OOPSLA'87*, SIGPLAN Notices, volume 22, numéro 12, pages 147–155, Orlando (Floride, USA), Octobre 1987. ACM Press.

[Mae88] P. MAES. « Issues in Computational Reflection ». Dans *Meta-Level Architectures and Reflection*, pages 21–35, North-Holland, 1988.

[Mar99] R. MARVIE. « CORBA Components : la proposition unifiée ». Dans *GDR Programmation*, Nantes, France, Mai 1999.

[MBVD+97] Vladimir MARANGOZOV, Luc BELLISSARD, Jean-Yves VION-DURY et Michel RIVEILL. « Connectors: a Key Feature for Building Distributed Components-Based Architectures ». Dans *Proc. of 2nd European Research Seminar and Advances in Distributed Systems (ERSADS'97)*, pages 246–251, Zinal, Mars 1997.

[MC93] P. MULET et P. COINTE. « Definition of a reflective kernel for a prototype-based language ». Dans *Proceedings of ISOTAS'93*, LNCS 742, pages 128–144. Novembre 1993, JSSST-JAIST, Springer-Verlag.

[McA95] J. MCAFFER. « Meta-level Programming with CODA ». Dans *Proceedings of ECOOP'95*, LNCS 952, pages 190–214. Août 1995, Springer Verlag.

[McA96] J. MCAFFER. « Meta-level Architecture Support for Distributed Objects ». Dans *Reflection'96*, pages 39–62, San Francisco (Californie, USA), Avril 1996.

[McC87] P.L. MCCULLOUGH. « Transparent Forwarding: First Steps ». Dans *Proceedings of OOPSLA'87*, SIGPLAN Notices, volume 22, numéro 12, pages 331–341, Orlando (Floride, USA), Octobre 1987. ACM Press.

[McH94] C. MCHALE. « *Synchronization in Concurrent Object-Oriented Languages: Expressive Power, Genericity and Inheritance* ». Thèse de Doctorat, Department of Computer Science, Trinity College, Dublin (Ireland), 1994.

[MDA01] « Model Driven Architecture - A Technical Perspective, OMG », 2001.

[MDE+95] J. MAGEE, N. DULAY, S. EISENBACH et J. KRAMER. « Specifying Distributed Software Architectures ». Dans *Fifth European Software Engineering Conference (ESEC'95)*, LNCS 989, pages 137–153, Barcelone (Espagne), Septembre 1995.

[MDK94] J. MAGEE, N. DULAY et J. KRAMER. « Regis: A Constructive Development Environment for Distributed Programs ». *IEE/IOP/BCS Distributed Systems Engineering*, volume 1, numéro 5, pages 304–312, Septembre 1994.

[Med97] N. MEDVIDOVIC. « A Classification and Comparison Framework for Software Architecture Description Languages ». Rapport Technique, Univ. of Irvine, Dept of Information and Computer Science, 1997.

[MG96] R. MONROE et D GARLAN. « Style-Based Reuse for Software Architectures ». Dans *Proceedings of the 1996 International Conference on Software Reuse*, Orlando (Floride, USA), Avril 1996.

[MGG96] Ph. MERLE, Ch. GRANSART et J.-M. GEIB. « CorbaScript and CorbaWeb: A Generic Object-Oriented Dynamic Environment upon CORBA ». Dans *Proceedings of TOOLS Europe'96*, Paris, France, Février 1996.

[MH99] V. MATENA et M. HAPNER. « Enterprise Java Beans Specification v1.1 - Public Draft », Mai 1999. Sun Microsystems.

[Mil92] R. MILNER. « The polyadic π-calculus: A tutorial ». Dans *Logic and Algebra of Specification*. 1992, Springer Verlag.

[MJD96] J. MALENFANT, M. JACQUES et F.-N. DEMERS. « A Tutorial on Behavioral Reflection and its Implementation », 1996.

[MK96] J. MAGEE et J. KRAMER. « Dynamic Structure in Software Architectures ». Dans *Proc. of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 3–14, Octobre 1996.

[MKL97] A. MENDHEKAR, G. KICZALES et J. LAMPING. « RG: A Case Study for Aspect-Oriented Programming ». Rapport Technique SPL97-009 P9710044, Xerox PARC, Palo Alto (Californie, USA), Février 1997.

[MKS89] J. MAGEE, J. KRAMER et M. SLOMAN. « Constructing Distributed Systems in Conic ». *IEEE Transactions on Software Engineering Journal*, volume 15, numéro 6, pages 663–675, Juin 1989.

[MLT+97] K. MENS, C. LOPES, B. TEKINERDOGAN et G. KICZALES. « Aspect-Oriented Programming ». Dans *ECOOP'97 Workshop Reader*, LNCS, pages 483–496. 1997, Springer Verlag.

[MM00] Raphäel MARVIE et Philippe MERLE. « Vers un modèle de composants pour CESURE - Le CORBA Component Model ». Rapport Technique 3, Projet RNRT 98 CESURE, Novembre 2000.

[MMA+95] H. MASUHARA, S. MATSUOKA, K. ASAI et A. YONEZAWA. « Compiling Away the Meta-Level in Object-Oriented Concurrent Reflective Languages Using Partial Evaluation ». Dans *Proceedings of OOPSLA'95*, SIGPLAN Notices, volume 30, numéro 10, pages 300–315, Austin (Texas, USA), Octobre 1995. ACM Press.

[MMG+01] Raphäel MARVIE, Philippe MERLE, Jean-Marc GEIB et Mathieu VADET. « OpenCCM : une plate-forme ouverte pour composants CORBA ». Dans *Actes de la seconde Conférence Française sur les Systèmes d'Exploitation (CFSE'2)*, 2001.

[MMSA+96] L. MOSER, P. MELLIAR-SMITH, D. AGARWAL, K. BUDHIA et C. LINGLEYPAPADOPOULOS. « Totem: A Fault-Tolerant Multicast Group Communication System ». *Communications of the ACM*, volume 39, numéro 4, pages 54–63, Avril 1996.

[MMW+92] H. MASUHARA, S. MATSUOKA, T. WATANABE et A. YONEZAWA. « Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently ». Dans *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'92)*, SIGPLAN Notices, pages 127–144. 1992, ACM Press.

[MOF97] « Meta Object Facility (MOF) Specification, Doc. Number 97.08.14 », Septembre 1997. `http://www.omg.org`.

[Moi01] S. MOISAN. « Réutilisation et générateurs de systèmes à base de connaissances : le *framework* BLOCKS ». *TSI*, volume 20, numéro 4, pages 529–553, 2001.

[MOR+96] N. MEDVIDOVIC, P. OREIZY, J.E. ROBBINS et R.N. TAYLOR. « Using Object-Oriented Typing to Support Architectural Design in the C2 Style ». Dans *Proceedings of the ACL SIGSOFT'96: Fourth Symposium on Foundations Software of Engineering (FSE)*, pages 24–32, Octobre 1996.

[MP98] J. MAGEE et D.E. PERRY, éditeurs. *Proceedings of Third International Software Architecture Workshop*, Novembre 1998.

[MPS93] S. MISHRA, L. PETERSON et R. SCHLICHTING. « Consul: a Communication Substrate for Fault-Tolerant Distributed Programs ». Dans *Distributed Systems Engineering*, pages 87–103, 1993.

[MRT99] N. MEDVIDOVIC, D.S. ROSENBLUM et R.N. TAYLOR. « A Language and Environment for Architecture-Based Software Development and Evolution ». Dans *Proceedings of 21st International Conference on Software Engineering (ICSE'99)*, pages 44–53, Mai 1999.

[MS00] « Microsoft, *About .Net*, White paper », Juin 2000. `http://wwwm.microsoft.com.whitepaper.asp`.

[Muk95] M. MUKHERJI. « *Specification of Multi-Object Coordination Schemes Using Coordinating Environments* ». Thèse de Doctorat, Department of Computer Science, Virginia Tech., Août 1995.

[MY93] S. MATSUOKA et A. YONEZAWA. « Analysis of inheritance anomaly in object-oriented concurrent programming languages ». Dans G. AGHA, P. WEGNER et A. YONEZAWA, éditeurs, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150, Cambridge (Massachusetts, USA), 1993. The MIT Press.

[Nau63] P. NAUR. « Revised Report on the Algorithmic Language Algol 60 ». *Comm. ACM*, volume 6, numéro 1, pages 1–17, 1963.

[ODP95] « ODP Reference Model: Overview », 1995. ITU-T | ISO/REC Recommendation X.901 | International Standard 10746-1.

[OMA97] « A Discussion of the Object Management Architecture », Janvier 1997. `http://www.omg.org`.

[OMG97a] « CorbaServices », 1997. `http://www.omg.org/library/csindx.html`.

[OMG97b] « *Event Management Service Specification*, Part 4 of CorbaServices Doc. Number 97.12.11 », Décembre 1997. `http://www.omg.org/library/csindx.html`.

[OMG97c] « *Relationship Service Specification*, Part 9 of CorbaServices Doc. Number 97.07.18 », Juillet 1997. `http://www.omg.org/library/csindx.html`.

[OMG98] « Object Management Group: *The common Object Request Broker Architecture* », Février 1998. http://www.omg.org.

[OMG99] « CORBA Components - Joint Revised Submission », Août 1999. OMG Management Group, BEA Systems Inc.

[OMG00] « Interoperable Naming Service Specification », Novembre 2000. http://www.omg.org.

[OMG01] « Common Object Request Broker Architecture (CORBA) v2.4.2 », Février 2001. http://www.omg.org.

[OOC98] « ORBacus Version 3.0. User and Programmer Guide », 1998. http://www.ooc.com.

[OW99] J. OVLINGER et M. WAND. « A Language for Specifying Traversals of Object Structures ». Dans *Proceedings of the 1999 Conference on Object-Oriented Programming Systems, Language and Applications (OOPSLA'99)*, SIGPLAN Notices, volume 34, numéro 10, Denver (Colorado, USA), Novembre 1999. ACM Press.

[Par96] H.V.D. PARUNAK. « Visualizing agent conversations: Using enhanced Dooley graphs for agent design and analysis ». Dans *Proceedings of the 2nd International Conference on Multiagent Systems*, pages 275–282. 1996, AAAI Press.

[Pas86] G.A. PASCOE. « Encapsulators: A New Software Paradigm in Smalltalk-80 ». Dans *Proceedings of OOPSLA'86*, pages 341–346, 1986.

[PDN89] R. PRIETO-DIAZ et J.M. NEIGHBORS. « Module Interconnection Languages ». *The Journal of Systems and Software*, volume 6, numéro 4, pages 307–334, Octobre 1989.

[Pet77] J.L. PETERSON. « Petri Nets ». Dans *ACM Computing Survey*, volume 9, numéro 3, pages 223–252, Septembre 1977.

[PFM01] A.-M. PINNA, M. FORNARINO et S. MOISAN. « Distributed Access Knowledge-Based System: Reified Service for Trace and Control ». Dans *International Symposium on Distributed Object Applicatons (DOA 2001)*, Rome, Italie, 2001. À paraître.

[Pur94] J.M. PURTILO. The POLYLITH Software Bus. Dans *ACM Transactions on Programming Languages and Systems*, volume 16, numéro 1, pages 151–174. Janvier 1994.

[PWG93] F. PACHET, F. WOLINSKI et S. GIROUX. « Spying as an Object-Oriented Programming Paradigm ». Dans *Proceedings of* TOOLS EUROPE *'93*, pages 109–118, 1993.

[RBM96] R. Van RENESSE, K. BIRMAN et S. MAFFEIS. « Horus: A Flexible Group Communication System ». Dans *Communication of the ACM*, SIGPLAN Notices, volume 39, numéro 4, pages 76–83. Avril 1996, ACM Press.

[RBP+91] J. RUMBAUGH, M. BLAHA, W. PREMERLANI, F. EDDY et W. LORENSON. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs (New Jersey, USA), 1991.

[Riv95] F. RIVARD. « Extension du compilateur Smalltalk : application à la paramétrisation de l'envoi de message ». Dans *JFLA'95*, 1995.

[Riv96] F. RIVARD. « Pour un lien d'instanciation dynamique dans les langages à classes ». Dans *JFLA'96*, Janvier 1996. INRIA - collection didactique.

[Roy97] P. Van ROY. « PIRATES : Méthodes et Outils pour la Programmation Répartie Transparente et Sûre », Juillet 1997. Proposition de Projet Convention de Recherche Région Wallonne.

[Rum87] J. RUMBAUGH. « Relations as Semantic Constructs in an Object-Oriented Language ». Dans *Proceedings of OOPSLA'87*, pages 466–481, Décembre 1987.

[San95] C. Souza Dos SANTOS. « *Un Mécanisme de Vues pour les Systèmes de Gestion de Bases de Données Objet* ». Thèse de Doctorat, Université de Paris-Sud, Paris (France), Novembre 1995.

[SDK+95] M. SHAW, R. DELINE, D.V. KLEIN, T.L. ROSS, D.M. YOUNG et G. ZELESNIK. Abstractions for Software Architecture and Tools That Supports Them. Dans *IEEE Transactions on Software Engineering (TSE)*, volume 21, numéro 4, pages 314–335. Avril 1995.

[SDZ96] M. SHAW, R. DELINE et G. ZELESNIK. « Abstraction and Implementations for Architectural Connections ». Dans *Proceedings of the Third International Conference on Configurable Distributed Systems*, Mai 1996.

[Ser99] M. SERRANO. « Wide classes ». Dans *Proceedings of ECOOP'99*, LNCS, pages 391–415, Lisbon, Portugal, Juin 1999. Springer-Verlag.

[SF96] J.M. SOBEL et D.P. FRIEDMAN. « An Introduction to Reflection-Oriented Programming ». Dans *Proceedings of Reflection'96*, San Francisco (Californie, USA), Avril 1996.

[Sha94] M. SHAW. « Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status ». Rapport Technique CMU-CS-94-107, School of Computer Science and Software Engineering Institute, Carnegie Mellon University, Pittsburgh (USA), Janvier 1994.

[Sha95] M. SHAW. « Architectural issues in software reuse: It's not just the functionnality, it's the packaging ». Dans *Proceedings of the IEEE Symposium on Software Reuse*, pages 3–6, Avril 1995.

[Sin00] M.P. SINGH. « Synthesizing Coordination Requirements for Heterogeneous Autonomous Agents ». *Autonomous Agents and Multi-Agent Systems*, volume 3, numéro 2, pages 107–132, 2000.

[Smi84] B.C. SMITH. « Reflection and semantics in Lisp ». Dans *Proceedings of ACM POPL'84*, pages 23–35, 1984.

[Smi90] B.C. SMITH. « What do you mean, meta? ». Dans *Workshop on Reflection and MetaLevel Architecture in OO Programming, ECOOP/OOPSLA'90*, Ottawa (Ontario, Canada), Octobre 1990.

[SML99] L. SEITER, M. MEZINI et K. LIEBERHERR. « Dynamic Component Gluing ». Dans *First International Symposium on Generative and Component-Based Software Engineering*, Septembre 1999.

[Sny86] A SNYDER. « Encapsulation and Inheritance in Object Programming Languages ». Dans *Proceedings of OOPSLA'86*, SIGPLAN Notices, volume 21, numéro 11, pages 38–45. Septembre 1986, ACM Press.

[SR92] B. SINGH et G.L. REIN. « Role Interaction Nets (RINs): A Process Description Formalism ». Rapport Technique, CT-083-92, MCC, 1992.

[SS94] L. STERLING et E. SHAPIRO. *The Art of Prolog*. MIT Press, 1994.

[Ste90] G.L. STEELE. *Common Lisp the Language*. Digital Press, 2nde édition, 1990.

[Str91] B. STROUSTRUP. *The C++ Programming Language*. Addison-Wesley, 2 édition, 1991.

[SUN97] « SUN MicroSystems, *Java Beans Specification*, version 1.01 », Juillet 1997.

[Tan95] C. TANZER. « Remarks on object-oriented modeling of associations ». Dans *Journal on Object-Oriented Programming*, pages 43–46, Février 1995.

[TBA89] A. TRIPATHI, E. BERGE et M. AKSIT. « An Implementation of the Object-Oriented Concurrent Programming Language SINA ». *Software Practice and Experience*, volume 19, numéro 3, pages 235–256, Mars 1989.

[VKC+99] R. VITTENBERG, I. KEIDAR, G.V. CHOCKLER et D. DOLEV. « Group Communication Specifications: A Comprehensive Study ». Rapport Technique MIT-LCS-TR-790, Computer Science Institute, The Hebrew University of Jerusalem and MIT, Septembre 1999.

[WF86] M. WAND et D.P. FRIEDMAN. « The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower ». Dans *Proceedings of the ACM Conference on LISP and Functional Programming*, ACM Press, pages 298–307, 1986.

[Wol96] A.L. WOLF, éditeur. *Proceedings of Second International Software Architecture Workshop (ISAW-2)*, Octobre 1996.

[WS99] I. WELCH et R. STROUD. « From *Dalang* to *Kava* - the Evolution of a Reflective Java Extension ». Dans *Proceedings of Meta-Level Architectures and Reflection*, LNCS 1616, pages 2–21, Saint-Malo (France), Juillet 1999. Springer Verlag.

[YS94] D.M. YELLIN et R.E. STROM. « Interfaces, Protocols, and the Semi-Automatic Construction of Software Adaptors ». Dans *OOPSLA'94*, pages 176–190, 1994.

[ZA00] A. ZUNINO et A. AMANDI. « Brainstorm/J: a Java Framework for Intelligent Agents ». Dans *Proceedings of the* 2^{nd} *Argentinian Symposium on Artificial Intelligence (ASAI'2000 -* 29^{th} *JAIIO)*, Tandil, Buenos Aires (Argentine), Septembre 2000.

MISE EN ŒUVRE DES INTERACTIONS EN ENVIRONNEMENTS DISTRIBUÉS, COMPILÉS ET FORTEMENT TYPÉS : LE MODÈLE MICADO

Résumé

La programmation orientée objet a déjà prouvé ses intérêts lors de la mise en œuvre d'applications complexes. Le développement des applications distribuées à l'aide de technologies objets est réalisable mais cela implique de gérer les communications entre les objets distants. Des outils tels que CORBA, RPC et Java RMI facilitent la mise en œuvre de la communication en masquant les accès réseaux. Cette maturation en termes de réseaux et de programmation par objets conduit aujourd'hui à une intensification du développement d'applications distribuées. Cette évolution des applications distribuées augmente le besoin de spécifier explicitement les sémantiques des communications et des interactions entre des objets.

Cependant, les outils mentionnés ci-dessus ne permettent pas d'exprimer les sémantiques des interactions entre des objets. Seuls quelques travaux vont dans le sens d'une expression et d'une gestion des interactions entre des objets distants indépendamment de leurs fonctionnalités intrinsèques. Cependant, il reste encore des travaux à faire sur la « sémantique » des interactions entre objets distants afin d'apporter encore plus de flexibilité, de facilité et une meilleure réutilisation lors de la mise en œuvre d'applications distribuées.

La solution avancée est la définition d'un modèle et d'une architecture distribuée de gestion des interactions entre objets distants dans les environnements de développement utilisés par le monde industriel, c'est-à-dire les environnements compilés, fortement typés et distribués. Elle est basée sur ISL (*Interaction Specification Language*), notre langage de description des interactions ainsi que sur un système de réécriture des comportements réactifs.

Mots-clefs : Architecture distribuée, Protocole à métaobjets, Environnements compilés et typés

Summary

Object oriented programming has already proved its interest to implement complex applications. Distributed applications can also be developed with object technologies but this implies to manage communications between remote objects. Tools such as CORBA, RPC and Java RMI facilitate the communication implementation by hiding network accesses. This maturation in terms of network and object programming is today sufficient to allow an important evolution for the groupware distributed applications. This evolution of distributed applications increases the need to specify explicitly the communication and interaction semantics between objects.

However the above-mentioned tools do not allow the expression of the semantics of interactions between objects. Few works deal with the way to express and to manage interactions between remote objects independently of their intrinsic behavior. However there are many works to do about the "semantics" of the interactions, i.e. inter object communications, between remote objects in order to bring more flexible, more reusable and more easy the implementation of distributed applications.

The proposed solution is the definition of a model to manage interactions between remote objects in environments used by the industry world, i.e. compiled, strongly typed and distributed environments. This solution is based upon ISL (*Interaction Specification Language*), our language to describe interactions and upon a rewriting system for reactive behaviors.

Keywords : Distributed Architecture, Meta-Object Protocol, Compiled and typed environments